OVERLAY TECHNIQUES FOR BLOCK STRUCTURED LANGUAGES

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of

MASTER OF TECHNOLOGY

By

RAJEEV SEONI

to the

DEPARTMENT OF ELECTRICAL ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

AUGUST 1980

CERTIFICATE

This is to certify that the work entitled, 'OVERLAY TECHNIQUES FOR BLOCK STRUCTURED LANGUAGES', has been carried out by Lt. Rajeev Seoni under our supervision and has not been submitted elsewhere for the award of a degree.

(V. RAJARAMAN)
Professor
Electrical Engineering Department
and
Computer Science Program
Indian Institute of Technology
Kanpur

(KESAV V. NORI)
Assistant Professor
Computer Science Program
Indian Institute of Technology, Kanpur

EE-1980-M-SEO-CVE

To

my Father and Mother,
and my sister Rashmi.

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my deep gratitude to Prof. Kesav V. Nori and Prof. V. Rajaraman (my thesis guides) for their help, guidance and encouragement.

I am thankful to Dr. S. Biswas and Mr. V.M. Malhotra, and to my colleage Mr. A.K. Dey, for their valuable suggestions.

My special thanks to Vandana for her corrective critisism of this thesis, and for all the help in proof-reading and preparation of slides.

My thanks to Mr. J.S. Rawat for typing and Mr. Ganga Ram for cyclostyling.

Rajeev Seoni

ABSTRACT

This thesis is concerned with the following two problems:

i) How can we decompose a large block structure program into smaller related components such that the resultant components can be used in Planned Overlay Schemes of memory management?

ii) What are the requirements of a separate compilation facility for block structured languages?

We have approached both these problems through PASCAL, an example block-structured language. As a test case, we have used the PASREL compiler and have obtained its decomposition that suits our purpose.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

In any computer system, efficient memory management is very important so as to obtain high efficiency of resource utilization and a satisfactory performance in the overall aims of the computer installation. The technique chosen for memory management should be closely dependant on the type of the work load of the computer and on the characteristics of the hardware available. Now, since the physical memory and even the virtual memory has some limits, it is possible that a program may be so large that it just cannot fit into the memory all at one time. Thus large programs cannot be run until and unless they are broken up into smaller parts in some way.

Our interest in this problem area arose because we were interested in transferring a large PASCAL compiler onto the local TDC-316 system.. A cross-compiler for PASCAL for PDP-11 running on the DEC system -10 was the starting point of this study. Unfortunately, the version of PASCAL on which this compiler could be cross-compiled was not available locally. We then became interested in segmenting the program into small segments so that it could fit in the limited memory resources of

TDC 316. Two problems that arose from this effort were:

i) How do we generally break a large block structured (PASCAL) program into small parts so as to run it on small machines using simple memory management techniques?

ii) How do we independantly compile procedures (arbitrarily nested) in a block structured language?

These two problems are tackled in this thesis.

1.1 Some Methods of Memory Management:

To execute programs too large for a certain computer system what is required, essentially, is to keep only those parts of the program in main memory which are required at that time. A reference to a part of the program not in main memory may cause the replacement of a part in memory by the referenced part.

At this point, it would be useful to clearly recognize that a program has its logical requirements of instruction code and data space. The logical requirement is met by mapping the instructions and data of the program into an address space, which may be

virtual or physical. Considering that the program itself is possibly constructed from several components with their own independent logical requirements, the basic problems in constructing the above mapping are relocation and linkage respectively.

Relocation causes a translation (shift) of some addresses used in the program so that the program (or even a component of a program) is consistent in its use of the address space with respect to a new origin.

Linkage problems are inherently concerned with the resolution of references across component boundaries (after the relocation problem has been tackled).

All memory management schemes are concerned with the above two issues. They differ only in the time, with respect to the execution of the program, at which the above problems are resolved.

Some of the methods of memory management available are, very briefly, as follows [2,3]:

(i) Planned Overlay: Overlaying is a technique where parts of a program are held on some external (or secondary) storage device and brought into main memory as required. In planned overlay, segments of the program are identified which need not be in main memory

together. The relationship of program segments also have to be planned in advance by the programmer. This method will be discussed in detail later.

In planned overlay, all relocation and linking is resolved before execution starts.

(ii) Dynamic Overlay: In this method no pre-planned overlay structure is required. As and when program segments are required, they are brought into main memory by explicit callsto the linking loader by the programmer. In certain cases, segments of the same program may be executed in parallel, in others, only serially.

(iii) Paging: In this scheme the main memory is divided into fixed length 'page frames', and each program into same length 'pages'. There are various sub-schemes for paging, which differ in when pages are brought into and when they are removed from main memory.

(iv) Segmenting: It is similar to paging, except that the programs are divided logically into variable length 'segments'. The segments may be further divided up into pages. Segmentation without paging is similar to Dynamic Serial Overlays.

1.2 Situations that are suitable for Planned Overlay:

In the case where the virtual memory is small or the physical memory is equal to the virtual memory, large programs will have to be broken up into an overlay structure to reduce their requirement of memory at any given time.

Programs that can be logically divided into major sections are well suited for planned overlay execution [2]. Also if the program structure follows well defined rules, planned overlay is suitable.

Planned overlay structures can be more efficient in terms of execution speeds compared to dynamic overlays because the linkage editor procedure permits direct references by one segment to values whose locations are identified by external symbols in another segment. There is no need to collect such values in a consolidated parameter list. Planned overlay optimizes the use of main memory, has lesser run-time overhead in comparison with Dynamic Overlay or Paging because there is no need for performing relocation, linkage or map table maintenance.

These advantages tend to diminish as the users' programs get more and more complex, particularly when

the logical selection of subprograms depends on the data being processed [2]. In this case, Dynamic Overlays seem better. A combination of both Planned and Dynamic Overlay structures may also be used. A module linked dynamically may itself operate in the planned overlay mode, and within a planned overlay program one may include dynamic overlaying.

1.3 Memory Requirements for Block Structured Languages:

Memory required by a program written in a block structured language can be classified as follows [4] :

i) Global data: This is permanently allocated for each program.

ii) Local data of procedures: This is usually allocated on the run-time stack on procedure call and deallocated on return from the procedure.

iii) Dynamically created data: This is usually allocated by a heap mechanism with some kind of garbage collection.

iv) Program code

Using Planned Overlay structures it is possible to overlay program code but not data. This is because data in the stack is dynamically created and referenced with respect to a base that is dynamically ascertained.

1.4 PASCAL Implementation :

The implementation of PASCAL involves the running of the PASCAL compiler, which itself is a large program. If the compiler is a one-pass type, it has to perform the whole lot of functions involved in compilation at one go. Thus it naturally becomes very large,and implementation of PASCAL on mini and micro computers is not possible due to memory constraints. If the compilation could be broken into a number of phases, then parts of the compiler can overlay each other, thereby reducing the total memory requirement. The phases could be scanning the input a number of times, each time performing a small task, or also having one phase produce an output which could be the input to the next phase. By increasing the number of phases, the compiler can be divided into smaller and smaller sections which may overlay each other, enabling the implementation of PASCAL on mini and micro computers.

1.5 Structure of the Thesis:

The first step in this thesis was the study of some tree - structured planned overlay systems,which is given in Chapter 2. In Chapter 3 are given a few algorithms which were developed and worked out for finding overlay tree structures, which did not give

satisfactory results, for block structured languages. Chapter 4 formalizes the basic problem in the construction of overlay trees for block-structured programs. Here, we apply the idea of strongly connected components of digraphs to the call graph of a program and obtain the minimally constrained Overlay Tree. A possible partition for the PASREL compiler, according to the algorithm which was implemented, is given in Chapter 5. Finally, in Chapter 6, the requirements for independant compilation facility for block structured languages to permit use of overlay techniques, are given. The last Chapter 7, contains the conclusions drawn from the work done for this thesis.

CHAPTER 2

OVERLAYING AND GENERAL REQUIREMENTS FOR TREE-STRUCTURED OVERLAYING LOADERS

Essentially, the technique of overlaying involves the division of large programs into smaller parts such that the parts are held on some secondary memory and brought into main memory only when they are required. Thus different parts of a program may occupy the same area of memory at different times. Generally, the routines of a program are grouped into a <u>permanent unit</u> and a number of <u>overlay units</u> (or <u>nodes</u>), and the available memory is divided into a <u>permanent area</u> and one or more <u>overlay areas</u> [1]. The permanent area holds the permanent unit as well as the non-overlaid data areas, which provides a communication area for overlay units occupying different overlay areas. Each overlay area holds, at any one time, one of a specified list of overlay units. This means that two units allocated to the same area cannot be in memory simultaneously.

2.1 <u>Overlaying:</u>

To enable overlaying, the assembler must provide pseudo - operations by which the programmer can indicate how the program is divided up, that is, which routines are in one overlay unit, which overlay units

occupy the same memory area and which routines are in the permanent area. This information has to be passed on by the assembler to the linkage editor and loader.

The implementation of the overlaying technique involves a number of stages [1] . In the first stage, each routine is compiled separately and each overlay unit is linked like a complete program. Its storage requirements are evaluated and cross-references filled in. The difference is that calls to other routines have their addresses flagged as relative to the start of their own overlay area (unknown at this time) or to the start of the permanent area.

The second stage determines the size of the overlay areas. The size of an area is obviously that of the largest overlay unit that will occupy it. Once the sizes of all overlay areas have been determined, memory can be allocated and the origin for each overlay area determined.

The last stage is loading. One-by-one, each overlay unit is processed, being relocated according to the origin previously calculated, and the resulting binary is outputted to the secondary store. At this stage, it is not necessary to load a unit into the memory area in which it will be executed.

Finally, the permanent area is set up, and a table is made which gives the secondary storage address of each overlay unit and its associated overlay area. The program which reads and writes overlay units (called the Overlay Handler) is incorporated into the permanent area by the usual library mechanism.

2.2 Requirements of a Tree-Structured Overlaying Loader:

Let us now consider the overlay facility available with the LINK-10 Linking Loader of the DEC system-10 [6]. The overlay program has a tree structure. The <u>nodes</u> of the tree are called <u>links</u>, each of which contains one or more program modules. The links are connected by <u>paths</u>.

The top node of the overlay tree is called the <u>root link</u>, and it contains the permanent overlay unit, that is, the main program, the Overlay Handler, the non-overlaid data areas and such procedures in the program which are required to be present in main memory throughout the execution of the program. Below the root link are the first-level links, each of which is connected to the root link by one path. The level of the links increases as we go further down away from the root link. A link at level n is connected by a path to exactly one link

(the father link) at level n-1. It is obvious that a link can have more than one downward path (to successor links), but only one upward path (to ancestor links).

An overlay tree structure with six links is shown in Fig. 1. The code in a given link can make reference only to memory in links along a direct upward or downward path, i.e. in a link which is vertically connected to it. Thus, the link C can reference memory in itself, in the root link A, or in its successor links D, E and F. A reference to memory in B from C would be illegal.

In the overlay tree, all nodes at any one level overlay each other. In the tree shown, B and C overlay each other, and only one amongst D, E and F can be in main memory at one time. One more type of reference that is not allowed, but may arise due to recursion, is a call from C to E if it is possible for F to call C. This is because, once F calls C and C in turn calls E, E would try to overlay F which has not yet finished execution.

Due to the restrictions in memory references, only one complete (at most) vertical path is required in

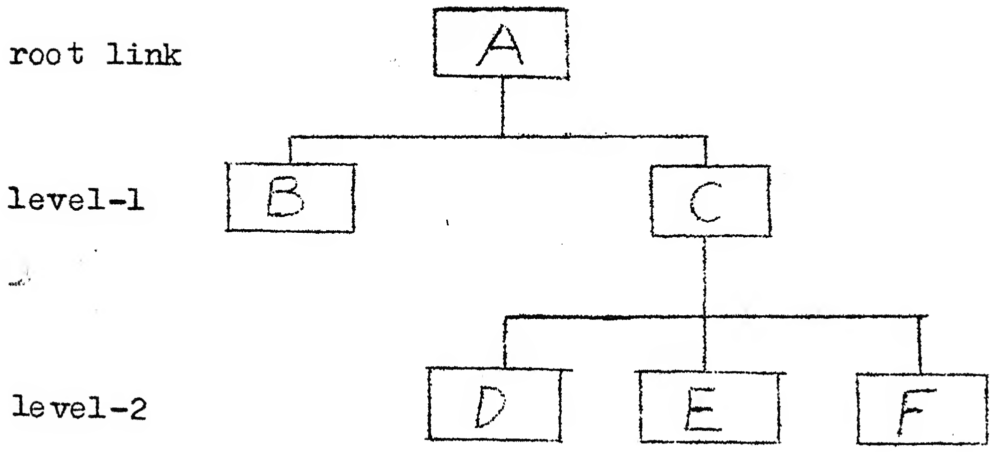


Fig. 1: An overlay tree structure

the virtual address space at any one time. The remaining links can be stored on a disk till they are required.

2.3 <u>Program to Supervise Overlays</u>:

There is a program which supervises the execution of an overlay program. Whenever a reference is made to a link which is not in memory, this program brings in the link, possibly overlaying one or more links already in memory. In the DEC system-10, this program is called the <u>Overlay Handler</u> [6]. The Overlay Handler is loaded into the root link, so that it is always in memory and can supervise overlaying operations from there.

Using the Overlay Handler, there are two ways of overlaying links during executions:

1. <u>Implicit overlaying</u>: A reference to a link not in memory implicitly calls the Overlay Handler to overlay one or more links with the required links.

CHAPTER 3

OVERLAY TECHNIQUES FOR BLOCK STRUCTURED PROGRAMS

The Block Structure Tree of a program is available to start with. Using the information of calls to and calls by each procedure, an overlay tree is to be constructed keeping in mind the requirements given in Chapter 2.

There are basically two ways in which the technique of overlaying can be used for reducing the memory requirements of a program:-

i) <u>Programmed (or dynamic) overlaying of nodes</u>: This method uses the facility of the Overlay Handler by which the programmer can explicitly cause the overlaying of overlay units (links). No re-structuring of the program is necessary, but the programmer has to insert calls to the Overlay Handler at the appropriate places in the program. That means that the programmer has to keep track of the amount of memory used up by the program at different times during execution and if it crosses a certain limit, he must overlay certain units. At this point he must be careful that he does not overlay a calling procedure, i.e., one which has been executed only partially and control has passed out of it due to a

call by that procedure. The programer will have to use the static link information to decide which overlay units will have to be brought in when a call is made to a procedure not in main memory.

This method is cumbersome and messy , and demands too much of work by the programmer. As far as restructuring and division of the program into overlay units is concerned, it is a trivial case.

(ii) Automatic overlaying: In this method, the Overlay Handler is called implicitly whenever a call is made to a procedure which is not in the main memory. The Overlay Handler then brings in the called unit, possibly overlaying a number of overlay units, according to some pre-planned overlay structure. This overlay structure is so planned that it takes care of all the requirements of the overlaying loader, or the Overlay Handler in DEC-10, like not overlaying a calling node etc.

So the main problem in using overlays boils down to re-structuring of the Block Structure Tree so that it conforms to the requirements of an overlaying loader. An algorithm which could give such an Overlay Tree would be very useful. Therefore a number of algorithms were developed and manually worked out for a large block structured program.

The best overlay tree is the one for which the total amount of memory required is the minimum, and also which uses memory for the least time during execution. Therefore, procedures which are rarely called should not be very high in the overlay tree because the higher a node, more is the time it spends in memory. Also, the overlay tree nodes should be as small as possible, in terms of memory required. A number of overlay trees may be possible for the same program. The programmer can select the one which has the minimum total memory requirement.

3.1 Heuristic Algorithms for Constructing Overlay Trees:

The basic methodology adopted in the algorithms developed is as follows:

The main program body was invariably put into the root node. All the procedures/functions declared in the main program (on level 1) were initially grouped together into one node, which is at level 1 of the Overlay Tree. In one of the algorithms, only those procedures which are declared in, as well as called by, the main program were put into this node. Then this node was split up into a number of brother nodes, all children of the same root node. All those procedures which call

each other have to be kept in one node. Those which do not call, and are not called by, any procedure in the node under consideration were seperated out into seperate nodes. A procedure which is called by a number of procedures in that node was shifted up into the parent node, or up into a new node at an intermediate level with all the calling procedures being kept in nodes which are children of this new node.

The above is repeated for all the procedures already placed in the overlay tree, i.e., all procedures declared in a certain procedure are initially grouped together into a child node, and then this node is split. But now one more thing has to be considered - a call backwards or upwards from the node under consideration, say from procedure A to procedure B. This sort of a call may cause problems, because the called procedure, B, may in turn call another procedure, say C, which may cause the overlaying of the first procedure, A. In this case, either the finally called (C) or the initially calling (A) procedure alongwith the procedures on the path have to be moved up into the node which has the backward called procedure (B). In one of the algorithms, the two paths were merged into one. One more possibility exists for a backward call- that the backward called

procedure does not fall on the vertical path to the root. In that case, it has to be moved up in the tree till it falls on the route, or else the two nodes (calling and called) have to be merged. After this step, all the calls to and from the shifted procedures have to be re-considered to eliminate all possibilities of a calling procedure being overlaid.

In the algorithms where only those procedures which are declared in, as well as called by, a procedure were initially grouped together, procedures which were not called were included in the first node from which they are called. This has an effect of pushing some procedures down in the overlay tree, which is good.

One of the algorithms developed took the block structure tree and started splitting/joining nodes from bottom up, using the same rules as the other algorithms discussed above.

It was seen that the algorithms were becoming very complicated because calls by procedures do not follow any set rules or pattern. A large number of possibilities had to be considered. Even then, it was seen that the overlay trees obtained did not help in reducing the total memory requirements of large programs because one of the

vertical paths invariably became very long compared to the others.

The possibility of duplicating procedures to seperate two paths was also considered. But ultimately it was felt that no automatic algorithm could be developed which would give an appreciable reduction in the total amount of memory required, or the time for which memory is required by a program. This was attributed to the numerous other factors involved.

Some of the factors responsible for making the construction of Overlay Trees more complicated are:

i) The number of times a procedure is called should affect its position in the Overlay Tree.

ii) The size of code of a procedure should also be considered.

iii) Average time of execution of a procedure has an affect on the time for which memory is required by it.

Considering all these points, it was felt that an automatic algorithm should just give the basic, essential division of a program (i.e. which procedures have to be in one node in all cases), alongwith all the calls information. Then it can be left to the programmer to

consider all the factors discussed and constructed an Overlay Tree. This sort of an algorithm is available if we adopt the Graph Theory approach, which is discussed in the next chapter.

CHAPTER 4

GRAPH THEORY APPLIED TO OVERLAY TREES

Before the application of Graph Theory to the construction of Overlay Trees is discussed, it would be useful to define those terms of Graph Theory which will be used in this thesis.

4.1 Definitions [7]:

(i) Graph: It is a finite set (V) together with a prescribed collection (E) of unordered pair of distinct elements.

e.g. : $V = \{1, 2, 3, 4, 5, 6\}$

$E = \{(1,2),\ (2,3),\ (3,4),\ (4,5),\ (5,6),\ (6,2),\ (6,4)\}$

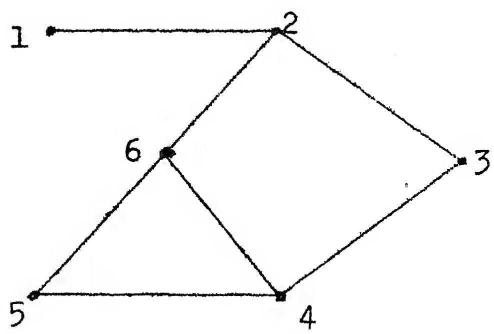


Fig. 2: An example of a Graph

(ii) Edge: Each unordered pair is called an edge, i.e., each element of E is an edge.

(iii) Vertex: Each element of the finite set (V) is called a Vertex. These are also known as Nodes of the graphs.

property if no larger subgraph contains it as a subgraph and has the property.

(xv) Strongly Connected (or Strong) Component: It is a maximal subgraph, of a digraph, in which every two points are mutually reachable.

4.2 An Algorithm to Determine the Strongly Connected Components of a Digraph:

A digraph and its strongly connected components are shown in Figure 3.

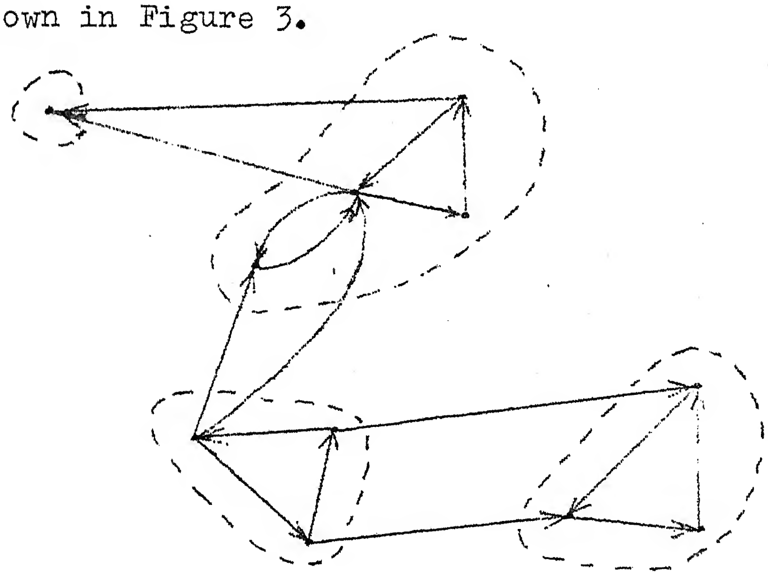


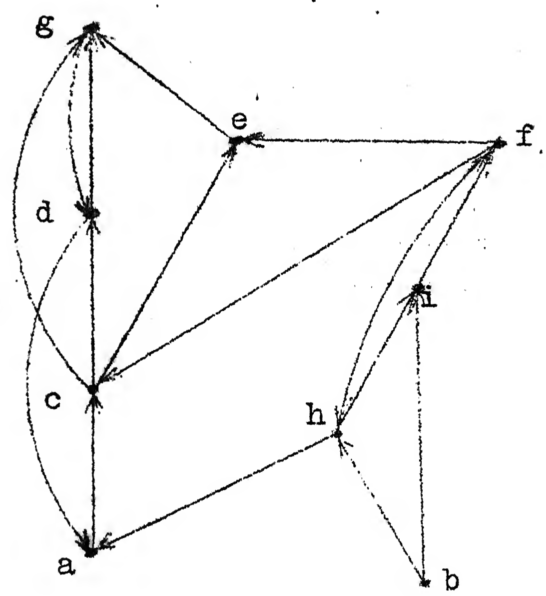
Figure 3: A digraph with its strongly connected components outlined by dashed lines.

Depth-first search can be applied to a digraph to determine the strongly connected components [5]. What is done in a depth-first search is this - one vertex (v) is "visited", than one of the edges from v (say, (v,w)) is

followed. If the vertex w has been previously visited, we return to v and choose another edge. If w has not been visited, we visit it and apply the process recursively to w. After all the edges leading out from v have been thus examined, we go back along the edge (u, v) that led us to v and continue exploring edges incident on u.

Let us consider what happens when we traverse the edges of a digraph G along their orientations during a depth-first search on G. We assign a serial number <u>nodeno</u>(x) to each vertex x the first time we visit it. If we encounter an edge (v, w) that has not been traversed, and w has not yet been visited, we mark this edge as a <u>tree edge</u>. If w has already been visited, then w may or may not be an ancestor of v. If w is an ancestor of v, then clearly $\text{nodeno}(w) \leq \text{nodeno}(v)$ and (v, w) is a <u>back edge</u>. If w is not an ancestor of v, and $\text{nodeno}(w) > \text{nodeno}(v)$ then w must be a descendant of v and the edge (v, w) is called a <u>forward edge</u>. If $\text{nodeno}(w) < \text{nodeno}(v)$ and w is neither an ancestor nor a descendant of v, then the edge (v, w) is called a <u>cross edge</u>.

Figure 4(a) shows a digraph G which is represented by its adjacency structure shown, and figure 4(b) shows



a:c
b:h,i
c:d,e,g
d:a,g
e:g
f:c,e,h
g:d
h:a,i
i:f

Fig. 4(a): A digraph G and its adjacency structure

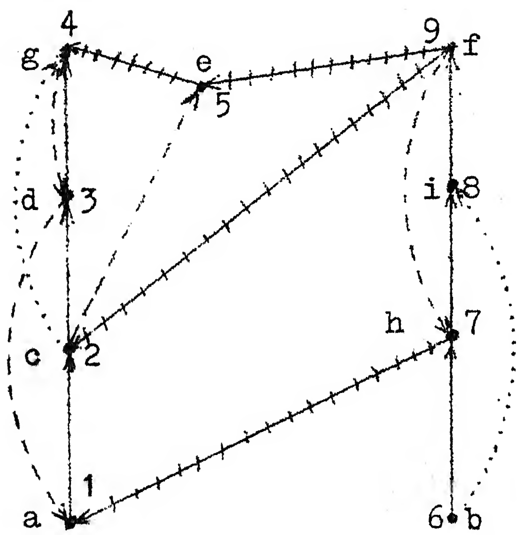


Fig. 4(b): Digraph G after a depth first search on it. The result consists of 2 trees (boldface), 3 back edges (dashed lines), 4 cross edges (crossed lines), and 2 forward edges (dotted lines).

the partitioning of the edges into four subsets as a result of the depth-first search. The numbers on vertices represent nodeno values.

Though the tree (or forest) generated by a depth - first search is not unique, it helps in determining the strongly connected components. Obviously, the forward edges can be ignored, since they do not affect strong connectivity. Also, both back and cross edges from v can only go to vertices x for which nodeno(v) $>$ nodeno(x). A little reasoning will show that if S is a strongly connected component of G, then the vertices of S define a tree which is a subgraph of the spanning forest.

To recognize the strongly connected components, we just have to identify the roots of the corresponding subtrees. To recognize these roots, we define enclblkno(v) to be the number of the smallest numbered vertex in the same strongly connected component as v that can be reached by following zero or more tree edges followed at most by one back edge or one cross edge. Thus, v is such a root if and only if nodeno(v) = enclblkno(v).

An algorithm, taken from [5], using the above facts and determining the strongly connected components of a digraph, given by its Adjacency structure Adj(x), is given below. We maintain a stack S.

```
procedure FINDCOMPS;
   begin
      i := 0;
      initialize S as an empty stack;
      for all vertices x in V do
            nodeno(x) := 0;
      for all vertices x in V do
            if nodeno(x) = 0 then STRONG(x)
   end;
procedure STRONG(v)
   begin
      i := i+1;
      nodeno(v) := i;
      enclblkno(v) := i;
      push v onto stack S;
      for all w in Adj(v) do
            if nodeno(w) = 0 then
               begin (*(v,w) is a tree edge*)
                 STRONG(w);
                 enclblkno(v) := minimum of (enclblkno(v),
                          enclblkno(w))
               end
             else if nodeno(w) < nodeno(v) then
             (*(v,w) is a back edge or a cross edge*)
```

```
            if w is on S then
             (*w is in the same strongly connected
               component as v, since w on S means
               there is a path from w to v *)
            enclblkno(v) :=    minimum of
               (enclblkno(v), nodeno(w));
       if enclblkno(v)   = nodeno(v)   then
              (* v is the root of a strongly
               connected component *)
               while x, the top vertex on S,
                    satisfies nodeno(x) >= nodeno(v) do
                       add x to the current strongly
                    connected component and delete x
                    from S.
end;
```

4.3 Call Graph and Overlay Tree Generation:

If each procedure/function in a program is thought of as a vertex and each call from one procedure to another as a directed edge, we have a digraph which is known as a call graph. Once we obtain the call graph, the first step towards generating the Overlay Tree would be to determine the strongly connected components of this graph. Now it is obvious that all the procedures in one strongly

connected component have to be kept in one node (or link) of the Overlay Tree. The last section shows that automatic determination of strongly connected components is quite simple. As pointed out in the last chapter, having generated this essential division of a program, we leave it to the programmer to construct the Overlay Tree considering the numerous other factors involved.

4.3.1 Call Graph Generation:

Generation of the call graph posed very interesting data structuring and procedure-name-table maintenance problems. We started with a simple lexical analyser which processed a subset of PASCAL (it was later modified to process the complete PASCAL).

We declared the following data structures:-

```
type
    PROCPTR    = ↑PROCNODE;
    LISTOFPROC = ↑NEXTPROC;
    NEXTPROC   = record
                    PROC : PROCPTR;
                    NEXT : LISTOFPROC
                 end;
```

```
PROCNODE  =  record
                NAME : ALPHA;
                LLINK, RLINK : PROCPTR;
                case ISPROC  : boolean of
                  true: (DECLPROC:PROCPTR;
                         CALLS, CALLEDBY,
                         STRONGCOMP:LISTOFPROC;
                         ONCSTACK: boolean;
                         NODENO, ENCLBLKNO: integer)
               end;
var
  DISPLAY: array [0..LEVMAX] of record
                                   PROCS :PROCPTR;
                                   CURRENT :PROCPTR
                                 end;
  TOP : 0.. LEVMAX;
```

The procedure-name-table is maintained in DISPLAY [TOP]. PROCS as an unbalanced binary tree at each declaration level. While processing a procedure/function declaration, a call is made to procedure ENTERPROC which creates a new PROCNODE, initializes all its fields and enters it in the appropriate place in the name-table. Then ENTERPROC also increments TOP, assigns the pointer to the procedure we have just entered to DISPLAY [TOP]. CURRENT, and initializes DISPLAY [TOP]. PROCS in anticipation of procedure declarations within this procedure.

On finally coming out of a procedure block, the DISPLAY [TOP] . CURRENT↑.DECLPROC is assigned the value of the pointer to the name-table (tree) of procedures declared in it, i.e., DISPLAY [TOP] . PROCS, and then TOP is decremented.

While processing the body of a procedure or function, whenever a call to another procedure or function is encountered, a call is made to procedure ENTERCALL which enters the call in both the called procedure (in field CALLEDBY) and the calling procedure (in field CALLS). At this point we realized that we were getting some errors due to the fact that a reference to a local variable in FACTOR was being entered as a call to a procedure of the same name at a lower level. Therefore, it was decided that all names of variables and constants declared should also be stored in the name-table, with the information that it is not a procedure being given by ISPROC.

Finally, we not only had the Call Graph, but also the Called By information and the static nesting of procedures in the input program.

4.3.2 <u>Determination of Strongly Connected Components</u>:

Procedures FINDCOMPS and STRONG (discussed in sec 4.2) were also implemented in the same program, to determine the strongly connected components of the Call Graph already generated. The listing of the program is attached (see Appendix A).

CHAPTER 5

APPLICATION TO A RECURSIVE DESCENT COMPILER

The algorithms developed in Chapter 3 were applied to the PASREL compiler. As mentioned in that chapter, an appreciable reduction in the total memory requirement was not obtained.

The results obtained on the application of the program discussed in Section 4.3 to the PASREL compiler are attached. The program gives the strongly connected components, the calls by each procedure (the Call Graph), the procedures which call a procedure, and the static nesting of procedures in the program (see Appendix B).

CHAPTER 6

CHANGES IN PASREL TO SUPPORT OVERLAYS

Since, in overlaying, the codes of procedures in a program may be overlaying each other, we would be interested in compiling each procedure (or a pre-planned group of procedures) seperately. Seperate compilation of procedures is possible in PASREL for those procedures which are declared at level 1, by using the M-option. We are interested in a facility that will allow seperate compilation of any procedure in a PASCAL program. For this purposes the scope rules of the language require that all the global user definitions with respect to the procedure of interest be available. To enable this, we have considered the two schemes given below.

(i) All the symbols or identifiers declared in a procedure are written out onto the secondary store, in a seperate file. In this way, all the symbols declared within different procedures will be available in different files on the secondary store. When a procedure body is to be compiled, those files which contain the symbols which can be referenced by this procedure (according to the rules of PASCAL) are brought into the main memory. One way of doing this is to include an option in PASREL, say S+, which constructs the global symbol table using the named files.

(ii) Only the procedure bodies should be declared EXTERN, so that the declarations within each procedure are treated as usual, i.e., the symbol table is maintained in the normal way.

CHAPTER 7

CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

After having developed and tried out a number of algorithms to generate Overlay Trees, we concluded that the problem was too complex, involving too many factors, for a completely automatic Overlay Tree Generator to be possible. The minimal requirements are met by the strongly connected components of the Call Graph, therefore this algorithm was implemented and tried out for the PASREL compiler. The results obtained are very encouraging. With all the calls between the strongly connected components available, it would be quite simple to generate the Overlay Tree manually by duplicating or merging nodes.

This algorithm is an instance of interesting language processing tasks. We must explore the possibilities of such models like Call Graphs for getting other properties of programs.

Though it is a basic requirement for overlaying, the implementation of seperate compilation of procedures on DEC system-10 would be a useful addition to the features of the compiler. It would make large programs

like the PASREL compiler, easier to handle for editing etc. At present, a minor alteration in a large program causes the editing of the entire file and the re-compilation of the whole program. With the seperate compilation facility, just one procedure may have to be re-compiled.

Our original aim was to suggest a way to break up large programs so that the PASREL compiler could be loaded and run on the local TDC-316 system. We have made such a suggestion, but the implementation of PASCAL on TDC-316 requires a lot of more work. We hope that some one would take up the challenge and go ahead with the task.

REFERENCES

1. D.W. Barron, Assemblers and Loaders (3rd edition), Macdonald and Jane's Computer Monographs, (1978).

2. S. Rosen (ed.), Programming systems and Languages, McGraw-Hill Book Company, (1967).

3. C.A.R. Hoare and R.H. Perrot (eds.), Operating Systems Techniques, Academic Press, (1972).

4. H.H. Nageli and R. Schoenberger, Preventing Storage Overflows in High-level Languages, Institut fur Informatik, ETH, Zürich, (1980).

5. E.M. Reingold, Jurg Nievergelt and Narsingh Deo, Combinatorial Algorithms - Theory and Practice, Prentice-Hall, Inc., (1977).

6. DEC System-10 Link Reference Manual, (1978).

7. Narsing Deo, Graph Theory with applications to Engineering and Computer Science, Prentice-Hall, Inc., (1974).

APPENDIX A

PROGRAM LISTING

This program carries out Syntax Analysis, generates the Call Gragh (alongwith the Called by and Static Nesting information), and determines its Strongly Connected Components.

```
const

  NORW = 39;
  AL = 10;
  LMAX = 132;
  NOERRMESS= ' ****  CONGRATS! YOU WIN !!NO ERRORS DETECTED';
  EOFMESS= ' NO INPUT FILE';
  EMES='  ERRORS DETECTED ';
  LEVMAX = 8;
  STMAX = 200;
  NOSP = 15;

type

  SYMBOL =
     (NUL,IDENT,INTNUM,REALNUM,PLUS,MINUS,TIMES,SLASH,POINTER,
        LPAREN,RPAREN,LBRACKET,RBRACKET,EQSYM,NESYM,LTSYM,EOFSYM,
        LESYM,GTSYM,GESYM,ASSIGN,COMMA,PERIOD,SEMICOLON,COLON,
        STRING,ANDSYM,ORSYM,NOTSYM,DIVSYM,MODSYM,BEGINSYM,ENDSYM,
        IFSYM,CONSTSYM,PACKEDSYM,
        THENSYM,ELSESYM,WHILESYM,DOSYM,REPEATSYM,UNTILSYM,
        TYPESYM,VARSYM,ARRAYSYM,OFSYM,FILESYM,RECORDSYM,
        FUNCSYM,PROCSYM,PROGSYM,INSYM,FORWARDSYM,SETSYM,
        LOOPSYM,EXITSYM,OTHERSYM,INITPROCSYm,EXTERNSYM,
        FORSYM,TOSYM,DOWNTOSYM,CASESYM,GOTOSYM,WITHSYM);

  ALPHA = packed array [1..AL] of char;

  SYMSET=set of SYMBOL;

  PROCPTR = ^ PROCNODE;
  LISTOFPROC = ^ NEXTPROC;
  NEXTPROC = record
                PROC : PROCPTR;
                NEXT : LISTOFPROC
             end;

  PROCNODE = record
                NAME : ALPHA;
                LLINK,RLINK : PROCPTR;
                case ISPROC : boolean of
                     true : (DECLPROC : PROCPTR;
                             CALLS,CALLEDBY,STRONGCOMP :
                             LISTOFPROC;
                             ONCSTACK : boolean;
                             NODENO,ENCLBLKNO : integer )
             end;

  STACK = array [0..STMAX] of PROCPTR;

var

  CH : char;

  SYM : SYMBOL;

  WORD : array [1..NORW] of ALPHA;

  WSYM : array [1..NORW] of SYMBOL;

  SSYM : array [char] of SYMBOL;

  LINE : packed array [1..LMAX] of char;

  CC,LL : 0..LMAX;

  ERRCOUNT:1..100;

  CONSTBEGSYm,SIMPTYBEGSym,SELECTSYS,TYPEBEGSYM,TYPDECL,
  DECLBEGSYM,STATBEGSYM,FACBEGSYM:SYMSET;
```

```
   DISPLAY : array[0..LEVMAX] of record
                                    PROCS : PROCPTR;
                                    CURRENT : PROCPTR
                                 end;

   TOP : 0..LEVMAX;

   PNAME : ALPHA;

   STDPR : array [1..NOSP] of ALPHA;

   NEWVARS : boolean;

   LNODE : PROCPTR;

   STPTR,OPPTR : integer;

   PSTACK,OPSTK : STACK;

procedure HALT;
   begin
   end;

procedure ERROR(N:integer);
   const
     ERRMES='ERROR  ';
   begin
     WRITELN(OUTPUT,ERRMES,N);
     WRITELN(TTY,ERRMES,N);
     ERRCOUNT:=ERRCOUNT+1
   end;


procedure NEXTCH;

   function CAPITAL(CH:char):char;
     begin CAPITAL:=CH;
       if ORD(CH)>140B then
          CAPITAL:=CHR(ORD(CH)-40B)
     end;

   begin
     if CC=LL then
       if EOF(INPUT) then  HALT
       else
        begin LL:=0; CC:=0;
          OUTPUT^:=' ';
          PUT(OUTPUT);
          while not(EOLN(INPUT)) do
           begin LL:=LL+1;
             LINE[LL]:=INPUT^;
             OUTPUT^:=INPUT^;
             PUT(OUTPUT);
             GET(INPUT)
           end;
          PUTLN(OUTPUT);
          LL:=LL+1;
          LINE[LL]:=' ';
          GET(INPUT)
        end;
     CC:=CC+1;
     CH:=CAPITAL(LINE[CC])
   end;

procedure GETSYM;
   var
     I,J,K : integer;
     A : ALPHA;

   function LETTER:boolean;
     begin
       if (ORD(CH)>=ORD('A')) and (ORD(CH)<=ORD('Z'
          LETTER:=true
       else LETTER:=false
     end;
```

```
function DIGIT:boolean;
   begin
     if (ORD(CH)>=ORD('0')) and (ORD(CH)<=ORD('9')) then
        DIGIT:=true
     else DIGIT:=false
   end;

begin
  while CH=' ' do NEXTCH;
  if LETTER then
   begin
     K:=0;
     while DIGIT or
     LETTER do
      begin
        if K<AL then
         begin K:=K+1;  A[K]:=CH
         end;
        NEXTCH
      end;
     while K<AL do
      begin K:=K+1;  A[K]:=' '
      end;
     I:=1;  J:=NORW;
     repeat
        K := (I+J) div 2;
        if A<=WORD[K] then J:=K-1;
        if A>=WORD[K] then I:=K+1
     until I>J;
     if I-1>J then SYM:=WSYM[K]
     else
      begin SYM:=IDENT; PNAME := A
      end
   end
  else
     if DIGIT then
      begin
        while DIGIT do NEXTCH;
        SYM:=INTNUM;
        if CH ='B' then NEXTCH
        else
         begin
           if CH='.' then
            begin NEXTCH;
              if CH='.' then CH:=':'
              else
                 if DIGIT then
                  begin SYM:=REALNUM;
                    while DIGIT do
                       NEXTCH
                  end
                 else
                  begin
                    SYM:=NUL;
                    ERROR(1);
                    GETSYM
                  end
            end;
           if (CH='E') or (CH='e') then
            begin NEXTCH;
              if (CH='+') or (CH='-') then NEXTCH;
              if DIGIT then
               begin SYM:=REALNUM;
                 while DIGIT do
                    NEXTCH
               end
              else
               begin
                 SYM:=NUL;
                 ERROR(2);
                 GETSYM
               end
            end
         end
      end
```

```
else
  if CH='''' then
  repeat
     NEXTCH;
     while CH<>'''' do NEXTCH;
     NEXTCH;
     SYM:=STRING
  until CH<>''''
  else
     if CH='<' then
      begin NEXTCH;
        if CH='>' then
         begin SYM:=NESYM;  NEXTCH
         end
        else
           if CH='=' then
            begin SYM:=LESYM;  NEXTCH
            end
           else SYM:=LTSYM
      end
     else
        if CH='>' then
         begin NEXTCH;
           if CH='=' then
            begin SYM:=GESYM;  NEXTCH
            end
           else SYM:=GTSYM
         end
        else
           if CH=':' then
            begin NEXTCH;
              if CH='=' then
               begin SYM:=ASSIGN;  NEXTCH
               end
              else SYM:=COLON
            end
           else
              if CH='.' then
               begin NEXTCH;
                 if CH='.' then
                  begin SYM:=COLON;  NEXTCH
                  end
                 else SYM:=PERIOD
               end
              else
                 if CH='(' then
                  begin NEXTCH;
                    if CH='*' then
                     begin NEXTCH;
                       repeat
                          while CH<>'*' do NEXTCH;
                          NEXTCH
                       until CH=')';
                       SYM:=NUL;
                       NEXTCH;
                       GETSYM
                     end
                    else
                       SYM:=LPAREN
                  end
                 else
                    if CH='%' then
                     begin
                       repeat
                          NEXTCH
                       until CH='\';
                       SYM := NUL;
                       NEXTCH;
                       GETSYM
                     end
```

```
                                        if (CH='+') or (CH='-') or
                                        (CH='*') or (CH='/') or
                                        (CH='^') or (CH='=') or (CH='(')
                                        or (CH=')') or
                                        (CH='[') or (CH=']') or (CH=',')
                                        or (CH=';') or
                                        (CH='#') then
                                         begin
                                           SYM:=SSYM[CH];  NEXTCH
                                         end
                                        else
                                         begin
                                           SYM:=NUL;
                                           ERROR(3);
                                           NEXTCH;
                                           GETSYM
                                         end;
   end;

function TESTSYM(LEX:SYMBOL):boolean;
   begin TESTSYM := LEX=SYM
   end;

function TESTSYMINSet(LEXSET:SYMSET):boolean;
   begin TESTSYMINSet := SYM in LEXSET
   end;


procedure TEST (S1,S2:SYMSET;N:integer);
   begin
     if not TESTSYMINSet(S1) then
      begin ERROR(N); S1 := S1 + S2;
         while not TESTSYMINSet(S1) do GETSYM
      end
   end;


procedure CHECKSYM(CSYM:SYMBOL;ERR:integer);
   begin
     if TESTSYM(CSYM) then GETSYM
     else ERROR(ERR)
   end;

procedure BSTINSERT(var INPROC : PROCPTR);
   var
     LPROC,LPROC1 : PROCPTR;
     LLEFT,ENTRYDONE : boolean;
   begin
     ENTRYDONE := false;
     LPROC := DISPLAY[ TOP ]. PROCS;
     if LPROC = nil
        then  DISPLAY[ TOP ]. PROCS := INPROC
     else
      begin
        repeat
           LPROC1 := LPROC;
           if LPROC ^. NAME <= INPROC ^. NAME
           then
            begin
              if LPROC ^. NAME = INPROC ^. NAME
              then
               begin ENTRYDONE := true; INPROC := LPROC
               end;
              LPROC := LPROC ^. RLINK; LLEFT := false
            end
           else
            begin
              LPROC := LPROC ^. LLINK; LLEFT := true
            end
        until LPROC = nil;
        if not ENTRYDONE
        then
           if LLEFT
              then  LPROC1 ^. LLINK := INPROC
           else  LPROC1 ^. RLINK := INPROC
      end
   end;
```

```
procedure ENTERVAR;
   var
      NEWVAR : PROCPTR;
   begin
      NEW( NEWVAR,false );
      with NEWVAR^ do
       begin
         NAME := PNAME; LLINK := nil; RLINK := nil
       end;
      BSTINSERT( NEWVAR )
   end;

procedure SIGNEDCONSt(FSYS:SYMSET);
   begin
      if ((TESTSYM(PLUS)) or (TESTSYM(MINUS))) then GETSYM;
      if TESTSYM(IDENT) then GETSYM
      else
         if ((TESTSYM(INTNUM)) or (TESTSYM(REALNUM))) then
            GETSYM
         else TEST([],FSYS,101)
   end;

procedure CONSTANTLISt( FSYS : SYMSET );
   begin
      if TESTSYM( STRING )
         then GETSYM
      else SIGNEDCONSt( FSYS + [COLON] );
      if SYM = COMMA
      then
       begin
         GETSYM;
         CONSTANTLISt( FSYS )
       end
   end;

procedure CONSTDEF(FSYS:SYMSET);
   begin
      TEST(CONSTBEGSYm,FSYS,102);
      if TESTSYM(STRING) then GETSYM
      else SIGNEDCONSt(FSYS)
   end;

procedure CONSTDECL(FSYS:SYMSET);
   begin
      TEST([IDENT],FSYS,103);
      while TESTSYM(IDENT) do
       begin ENTERVAR; GETSYM;
         if TESTSYM(EQSYM) then
          begin GETSYM;
            CONSTDEF(FSYS+[SEMICOLON]);
            CHECKSYM(SEMICOLON,5)
          end
         else ERROR(4)
       end;
      TEST([IDENT]+FSYS,[],104)
   end;

procedure IDENTLIST(FSYS:SYMSET);
   begin
      TEST([IDENT],FSYS,606);
      if TESTSYM(IDENT) then
       begin
         if NEWVARS then ENTERVAR;
         GETSYM
       end;
      while TESTSYM(COMMA) do
       begin GETSYM;
         IDENTLIST(FSYS+[COMMA])
       end
   end;
```

```
procedure SIMPLETYPE(FSYS:SYMSET);
   begin
     TEST(SIMPTYBEGSym,FSYS,110);
     if TESTSYMINSet(SIMPTYBEGSym) then
      begin
        if TESTSYM(STRING) then
         begin GETSYM;
           CHECKSYM(COLON,20); CHECKSYM(STRING,21)
         end
        else
           if TESTSYM(LPAREN) then
            begin GETSYM; NEWVARS := true;
              IDENTLIST(FSYS+[RPAREN]);
              NEWVARS := false;
              CHECKSYM(RPAREN,22)
            end
           else
            begin SIGNEDCONSt(FSYS+[COLON]);
              if TESTSYM(COLON) then
               begin GETSYM; SIGNEDCONSt(FSYS)
               end
            end
      end
   end;


procedure TYPEDEF(FSYS:SYMSET);
  forward;


procedure ARRAYTYPE(FSYS:SYMSET);
   begin TEST([LBRACKET],FSYS,23);
     if TESTSYM(LBRACKET) then
      begin GETSYM;
        SIMPLETYPE(FSYS+[COMMA,RBRACKET]);
        while TESTSYM(COMMA) do
         begin SIMPLETYPE(FSYS+[COMMA,RBRACKET])
         end;
        if TESTSYM(RBRACKET) then GETSYM
        else ERROR(24);
        if TESTSYM(OFSYM) then GETSYM
        else ERROR(25);
        TYPEDEF(FSYS)
      end
   end;


procedure FIELDLIST(FSYS:SYMSET);
   begin
     while TESTSYM( IDENT ) do
      begin
        IDENTLIST( FSYS + [COLON] );
        CHECKSYM( COLON,951 );
        TYPEDEF( FSYS + [SEMICOLON,ENDSYM,CASESYM] );
        if TESTSYM( SEMICOLON ) then GETSYM
      end;
     if TESTSYM( CASESYM ) then
      begin
        GETSYM;
        CHECKSYM( IDENT,952 );
        if TESTSYM( COLON ) then
         begin GETSYM; CHECKSYM( IDENT,953 )
         end;
        CHECKSYM( OFSYM,954 );
        loop
           CONSTANTLISt( FSYS + [COLON] );
           CHECKSYM( COLON,955 );
           CHECKSYM( LPAREN,956 );
           FIELDLIST( FSYS + [RPAREN] );
           CHECKSYM( RPAREN,957 );
        exit if SYM # SEMICOLON;
           GETSYM;
        end
      end
   end;
```

```
procedure RECARR(FSYS:SYMSET);
   begin
     if TESTSYM(RECORDSYM) then
      begin GETSYM;
        FIELDLIST(FSYS+[ENDSYM]);
        CHECKSYM(ENDSYM,28)
      end
     else
      begin CHECKSYM(ARRAYSYM,29);
        ARRAYTYPE(FSYS)
      end
   end;


procedure SETYPE(FSYS:SYMSET);
   begin CHECKSYM(OFSYM,30);
     SIMPLETYPE(FSYS)
   end;


procedure TYPEDEF;
   begin
     TEST(TYPEBEGSYM,FSYS,112);
     if TESTSYMINSet(TYPEBEGSYM) then
      begin
        if TESTSYM(SETSYM) then
         begin GETSYM;  SETYPE(FSYS)
         end
        else
           if TESTSYM( FILESYM ) then
            begin GETSYM;CHECKSYM( OFSYM,930 );
              TYPEDEF( FSYS )
            end
           else
              if TESTSYM(PACKEDSYM) then
               begin GETSYM; TYPEDEF( FSYS )
               end
              else
                 if TESTSYM(POINTER) then
                  begin GETSYM; CHECKSYM(IDENT,31)
                  end
                 else
                    if (TESTSYMINSet( [RECORDSYM,ARRAYSYM]))
                       then RECARR(FSYS)
                    else SIMPLETYPE(FSYS);
        TEST(FSYS,[],113)
      end
   end;

procedure TYPEDECL(FSYS:SYMSET);
   begin
     TEST([IDENT],FSYS,103);
     while TESTSYM(IDENT) do
      begin ENTERVAR; GETSYM;
        CHECKSYM(EQSYM,4);
        TYPEDEF(FSYS+[SEMICOLON]);
        CHECKSYM(SEMICOLON,5)
      end;
     TEST([IDENT]+FSYS,[],104)
   end;


procedure VARDECL(FSYS:SYMSET);
   begin
     NEWVARS := true;
     repeat
        IDENTLIST(FSYS+[COLON]);
        CHECKSYM(COLON,32);
        TYPEDEF(FSYS+[SEMICOLON]);
        CHECKSYM(SEMICOLON,33)
     until (not TESTSYM(IDENT))  and not TESTSYMINSet(TYPDECL);
     NEWVARS := false
   end;
```

```
procedure ENTERPROC;
    var
      NEWPROC : PROCPTR;
    begin
      NEW( NEWPROC,true );
      with NEWPROC ^ do
       begin
         NAME := PNAME;  LLINK := nil;  RLINK := nil;
         CALLEDBY := nil;  ENCLBLKNO := 0; NODENO := 0;
         DECLPROC := nil;  CALLS := nil ; STRONGCOMP := nil;
         ONCSTACK := false
       end;
      BSTINSERT(NEWPROC);
      TOP := TOP + 1;
      DISPLAY[ TOP ]. PROCS := nil;
      DISPLAY[ TOP ]. CURRENT := NEWPROC
    end;


procedure ENTERSTDPROcs;
    var
      I : integer;
      STDPROC : PROCPTR;
    begin
      I := 1;
      DISPLAY[ TOP ]. PROCS := nil;
      repeat
         NEW( STDPROC,true );
         with STDPROC ^ do
          begin
            NAME := STDPR[I];  LLINK := nil;  RLINK := nil;
            CALLEDBY := nil;  DECLPROC := nil; ENCLBLKNO := 0;
            CALLS := nil; STRONGCOMP := nil; NODENO := 0;
            ONCSTACK := false
          end;
         if I = 1 then DISPLAY[ 1 ]. CURRENT := STDPROC;
         BSTINSERT(STDPROC);
         I := I + 1
      until I > 14
    end;

function SEARCHPROC(SNAME : ALPHA ) : PROCPTR;
    var
      LPROC : PROCPTR;
      LTOP : integer;
      FLAG : boolean;
    begin
      LTOP :=TOP;
      repeat
         LPROC := DISPLAY[ LTOP ]. PROCS;
         FLAG := LPROC # nil;
         while FLAG do
          begin
            if LPROC ^. NAME < SNAME
               then LPROC := LPROC ^. RLINK
            else
               if LPROC ^. NAME = SNAME
                  then FLAG := false
               else LPROC := LPROC ^. LLINK;
            if FLAG then FLAG := LPROC # nil
          end;
         LTOP := LTOP - 1;
         if LPROC # nil then FLAG := LPROC ^. NAME = SNAME
      until (LTOP < 0) or FLAG;
      if FLAG then FLAG := LPROC ^. ISPROC;
      if FLAG then SEARCHPROC := LPROC
      else SEARCHPROC := nil
    end;
```

```
procedure ENTERCALL;
   var
     LLP : LISTOFPROC;
     FLAG : boolean;
     CALLEDPROC : PROCPTR;
   begin
     LLP := DISPLAY[ TOP ]. CURRENT ^. CALLS;
     FLAG := LLP # nil;
     while FLAG do
      begin
        if LLP ^. PROC ^. NAME = PNAME then FLAG := false;
        if FLAG then
         begin LLP := LLP ^. NEXT; FLAG := LLP # nil
         end
      end;
     if LLP = nil
     then
      begin
        CALLEDPROC := SEARCHPROC( PNAME );
        if CALLEDPROC # nil (* CALLEDPROC=NIL INDICATES IT IS A
                                VARIABLE*)
        then
         begin
           NEW( LLP );
           LLP ^. PROC := CALLEDPROC;
           LLP ^. NEXT := DISPLAY[ TOP ]. CURRENT ^. CALLS;
           DISPLAY[ TOP ]. CURRENT ^. CALLS := LLP;
           NEW( LLP );
           LLP ^. PROC := DISPLAY[ TOP ]. CURRENT;
           LLP ^. NEXT := CALLEDPROC ^. CALLEDBY;
           CALLEDPROC ^. CALLEDBY := LLP
         end
      end
   end;

procedure EXPRESSION(FSYS:SYMSET);
  forward;

procedure EXPLIST(FSYS:SYMSET);
   begin
     EXPRESSION(FSYS+[COMMA,COLON]);
     if TESTSYMINSet( [COMMA,COLON] ) then
      begin GETSYM;
        EXPLIST(FSYS)
      end;
     TEST(FSYS,[],601)
   end;

procedure SELECTOR(FSYS:SYMSET);
   begin
     if TESTSYMINSet(SELECTSYS) then
      begin
        if TESTSYM(LBRACKET) then
         begin GETSYM;
           EXPLIST(FSYS+[RBRACKET]);
           CHECKSYM(RBRACKET,9)
         end
        else
           if TESTSYM(PERIOD) then
            begin GETSYM;
              CHECKSYM(IDENT,10)
            end
           else
              if TESTSYM(POINTER) then GETSYM;
        SELECTOR(FSYS)
      end
   end;

procedure FUNORVAR(FSYS:SYMSET);
   begin SELECTOR(FSYS+[LPAREN]);
     if TESTSYM(LPAREN) then
      begin GETSYM;
        EXPLIST(FSYS+[RPAREN]);
        CHECKSYM(RPAREN,11)
      end
   end;
```

```
procedure FACTOR(FSYS:SYMSET);
   begin
     TEST(FACBEGSYM,FSYS,107);
     if TESTSYMINSet(FACBEGSYM) then
      begin
        if TESTSYM(IDENT) then
         begin ENTERCALL; GETSYM; FUNDRVAR(FSYS)
         end
        else
           if (TESTSYMINSet( [INTNUM,REALNUM,STRING])) then
              GETSYM
           else
              if TESTSYM(NOTSYM) then
               begin GETSYM; FACTOR(FSYS)
               end
              else
                 if TESTSYM(LPAREN) then
                  begin GETSYM;
                    EXPRESSION(FSYS+[RPAREN]);
                    CHECKSYM( RPAREN,12)
                  end
                 else
                    if TESTSYM(LBRACKET) then
                     begin
                       GETSYM;
                       if not (TESTSYM(RBRACKET)) then
                          EXPLIST(FSYS+[RBRACKET]);
                       CHECKSYM(RBRACKET,13)
                     end
      end;
     TEST(FSYS,[],108)

   end;


procedure TERM(FSYS:SYMSET);
   begin FACTOR(FSYS+[TIMES,SLASH,DIVSYM,MODSYM,ANDSYM]);
     while (TESTSYMINSet( [TIMES,SLASH,DIVSYM,MODSYM,ANDSYM]))
     do
      begin GETSYM; FACTOR(FSYS+[TIMES,SLASH,DIVSYM,MODSYM,
                                  ANDSYM])
      end
   end;


procedure SIMEXP(FSYS:SYMSET);
   begin
     if (TESTSYMINSet( [PLUS,MINUS])) then GETSYM;
     TERM(FSYS+[PLUS,MINUS,ORSYM]);
     while (TESTSYMINSet( [PLUS,MINUS,ORSYM])) do
      begin GETSYM; TERM(FSYS+[PLUS,MINUS,ORSYM])
      end
   end;


procedure EXPRESSION;
   begin SIMEXP(FSYS+[EQSYM,NESYM,LTSYM,LESYM,GTSYM,GESYM,INSYM]
     );
     if (TESTSYMINSet( [EQSYM,NESYM,LTSYM,LESYM,GTSYM,GESYM,
                         INSYM])) then
      begin GETSYM;
         SIMEXP(FSYS)
      end
   end;


procedure PUSH(var STNODE : PROCPTR;var STK : STACK ; var PTR :
               integer);
   begin
     PTR := PTR + 1;
     if PTR > STMAX
        then ERROR( 300 )
     else STK[PTR] := STNODE
   end;
```

```
procedure POP(var STNODE : PROCPTR;var STK : STACK ; var PTR :
                    integer);
   begin
     if PTR = 0
        then ERROR( 301 )
     else
      begin STNODE := STK[PTR]; PTR := PTR - 1
      end
   end;

function STACKEMPTY(PTR : integer) : boolean;
   begin
     if PTR = 0 then STACKEMPTY := true
     else STACKEMPTY := false
   end;

procedure FINDCOMPS;
   var
     COMPSTK : STACK;
     CPTR,I : integer;

   function MIN( L,M : integer ) : integer;
      begin
        if L < M then MIN := L
        else MIN := M
      end;

   procedure ENTERCOMP( var LPROC,LLPROC : PROCPTR );
      var
        COMP : LISTOFPROC;
      begin
        NEW( COMP );
        COMP ^. PROC := LLPROC;
        COMP ^. NEXT := LPROC ^. STRONGCOMP;
        LPROC ^. STRONGCOMP := COMP
      end;

   procedure STRONG( var LPROC : PROCPTR );
      var
        LLPROC : PROCPTR;
        LLP : LISTOFPROC;
        FLAG : boolean;
      begin
        I := I + 1;
        LPROC ^. NODENO := I;
        LPROC ^. ENCLBLKNO := I;
        PUSH( LPROC,COMPSTK,CPTR );
        LPROC ^. ONCSTACK := true;
        LLP := LPROC ^. CALLS;
        while LLP # nil do
         begin
           LLPROC := LLP ^. PROC;
           if LLPROC ^. NODENO = 0 then
            begin STRONG( LLPROC );
              LPROC ^. ENCLBLKNO := MIN( LPROC^.ENCLBLKNO,
                                         LLPROC^.ENCLBLKNO )
            end
           else
              if LLPROC^.NODENO < LPROC^.NODENO then
                 if LLPROC^.ONCSTACK then
                    LPROC^.ENCLBLKNO := MIN( LPROC^.ENCLBLKNO,
                                             LLPROC^.NODENO );
           LLP := LLP ^. NEXT
         end;
        if LPROC^.ENCLBLKNO = LPROC^.NODENO then
         begin
           POP( LLPROC,COMPSTK,CPTR );
           LLPROC ^. ONCSTACK := false;
           FLAG := LLPROC ^. NODENO >= LPROC ^. NODENO;
           while FLAG do
            begin
              ENTERCOMP( LPROC,LLPROC );
              if STACKEMPTY( CPTR )
                 then FLAG := false
              else
```

```
                    POP( LLPROC,COMPSTK,CPTR );
                    LLPROC ^. ONCSTACK := false;
                    FLAG := LLPROC ^. NODENO >= LPROC ^.NODENO
                  end
               end;
             if LLPROC ^. NODENO < LPROC ^. NODENO then
               begin PUSH( LLPROC,COMPSTK,CPTR );
                 LLPROC ^. ONCSTACK := true
               end
            end
         end;

      procedure TRAVERSETRee;
         begin
           while LNODE # nil do
             begin PUSH(LNODE,PSTACK,STPTR); LNODE := LNODE ^.
               LLINK
             end;
           while not STACKEMPTY(STPTR) do
             begin
               POP(LNODE,PSTACK,STPTR);
               if LNODE ^. ISPROC then
                begin
                  if LNODE ^. NODENO = 0 then STRONG( LNODE );
                  if LNODE ^. DECLPROC # nil then PUSH(LNODE,DPSTK,
                                                  DPPTR)
                end;
               LNODE := LNODE ^. RLINK;
               while LNODE # nil do
                begin PUSH(LNODE,PSTACK,STPTR); LNODE := LNODE ^.
                  LLINK
                end
             end
         end;
      begin % FINDCOMPS \
        STPTR := 0; DPPTR := 0; CPTR := 0; I := 0;
        LNODE := DISPLAY[ 0 ]. PROCS;
        STRONG( LNODE );
        loop
           LNODE := LNODE ^. DECLPROC;
           TRAVERSETRee;
        exit if STACKEMPTY( DPPTR );
           POP( LNODE,DPSTK,DPPTR )
        end
      end;


procedure WRITEPINFO;

   procedure WRITEPTREE;

      procedure WRITENODE;
         var
           COUNT : integer;
         begin
           if (LNODE ^. CALLS = nil) and (LNODE ^. CALLEDBY =
                                             nil)

           then
            begin
              WRITELN;
              WRITELN(LNODE ^. NAME,'  is not called in this program.'
                     )
            end
           else
            begin
              WRITELN;
              WRITELN( LNODE ^. NAME );
              if LNODE ^. STRONGCOMP # nil then
               begin
                 WRITELN('   It is the root of a strongly connected componen
                        );
                 WRITELN('      which consists of the following procedure(s)
                        );
```

```
            COUNT := 1;
            while LNODE ^. STRONGCOMP # nil do
             begin
               WRITE('    ',LNODE ^. STRONGCOMP ^. PROC ^.
                      NAME );
               LNODE ^. STRONGCOMP := LNODE ^. STRONGCOMP ^
               . NEXT;
               COUNT := COUNT + 1;
               if COUNT > 5 then
                begin WRITELN; COUNT := 1
                end
             end;
            WRITELN
          end;
         if LNODE ^. CALLS # nil
         then
          begin
            WRITELN('   It calls the following procedures : '
                    );
            COUNT := 1;
            while LNODE ^. CALLS # nil do
             begin
               WRITE( '    ',LNODE ^. CALLS ^. PROC ^. NAME
                      );
               LNODE ^. CALLS := LNODE ^. CALLS ^. NEXT;
               COUNT := COUNT + 1;
               if COUNT > 5 then
                begin WRITELN; COUNT := 1
                end
             end;
            WRITELN
          end;
         if LNODE ^. CALLEDBY # nil
         then
          begin
            WRITELN('   It is called by the following procedures : '
                    );
            COUNT := 1;
            while LNODE ^. CALLEDBY # nil do
             begin
               WRITE( '    ',LNODE ^. CALLEDBY ^. PROC ^.
                      NAME );
               LNODE ^. CALLEDBY := LNODE ^. CALLEDBY ^.
               NEXT;
               COUNT := COUNT + 1;
               if COUNT > 5 then
                begin WRITELN; COUNT := 1
                end
             end;
            WRITELN
          end
       end
    end;

begin (* WRITEPTREE *)
  while LNODE # nil do
   begin PUSH(LNODE,PSTACK,STPTR); LNODE := LNODE ^.
     LLINK
   end;
  while not STACKEMPTY(STPTR) do
   begin
     POP(LNODE,PSTACK,STPTR);
     if LNODE ^. ISPROC then
      begin WRITENODE;
        if LNODE ^. DECLPROC # nil then PUSH(LNODE,DPSTK,
                                          DPPTR)
      end;
     LNODE := LNODE ^. RLINK;
     while LNODE # nil do
      begin PUSH(LNODE,PSTACK,STPTR); LNODE := LNODE ^.
        LLINK
      end
   end
end;
```

```
    begin (* WRITEPINFO *)
      PAGE;
      WRITELN;
      WRITELN('  LIST OF PROCEDURES AND FUNCTIONS IN THE PROGRAM'
              );
      WRITELN(' ----------------------------------------------'
              );
      WRITELN('      with static nesting and calls information');
      STPTR := 0; DPPTR := 0;
      LNODE := DISPLAY[ 0 ]. PROCS;
      WRITEPTREE;
      while not STACKEMPTY(DPPTR) do
       begin
         POP(LNODE,DPSTK,DPPTR);
         WRITELN;
         WRITELN('  Following procedures are declared in ',LNODE^
                 .NAME);
         WRITELN(' ---------------------------------------------'
                 );
         LNODE := LNODE ^. DECLPROC;
         WRITEPTREE
       end
    end;

procedure STATEMENT(FSYS:SYMSET);
  forward;

procedure STATLIST(FSYS:SYMSET);
    begin
      STATEMENT(FSYS+[SEMICOLON]);
      if TESTSYM(SEMICOLON) then
       begin GETSYM;
         STATLIST(FSYS)
       end;
      TEST(FSYS,[],600)
    end;


procedure IFSTAT(FSYS:SYMSET);
    begin EXPRESSION(FSYS+[THENSYM]);
      CHECKSYM(THENSYM,14);
      STATEMENT(FSYS+[ELSESYM]);
      if TESTSYM(ELSESYM) then
       begin GETSYM; STATEMENT(FSYS)
       end
    end;


procedure WHILESTAT(FSYS:SYMSET);
    begin EXPRESSION(FSYS+[DOSYM]);
      CHECKSYM(DOSYM,15);
      STATEMENT(FSYS)
    end;


procedure REPEATSTAT(FSYS:SYMSET);
    begin STATLIST(FSYS+[UNTILSYM]);
      CHECKSYM(UNTILSYM,16);
      EXPRESSION(FSYS)
    end;


procedure OTHERSTAT(FSYS:SYMSET);
    begin SELECTOR(FSYS+[ASSIGN]);
      if TESTSYM(ASSIGN) then
       begin GETSYM; EXPRESSION(FSYS)
       end
      else
       begin
         ENTERCALL;
         if TESTSYM(LPAREN) then
          begin GETSYM;
            EXPLIST(FSYS+[RPAREN]);
            CHECKSYM(RPAREN,17)
          end
       end
    end;
```

```
procedure FORSTAT( FSYS : SYMSET );
   begin
     CHECKSYM( IDENT,601 );
     CHECKSYM( ASSIGN,602 );
     EXPRESSION( FSYS + [TOSYM,DOWNTOSYM] );
     if TESTSYMINSet( [TOSYM,DOWNTOSYM] )
        then GETSYM
     else ERROR( 603 );
     EXPRESSION( FSYS + [DOSYM] );
     CHECKSYM(DOSYM,603);
     STATEMENT( FSYS )
   end;

procedure WITHSTAT( FSYS : SYMSET );
   begin
     repeat
        if TESTSYM( COMMA ) then GETSYM;
        CHECKSYM( IDENT,605 );
        SELECTOR( FSYS + [COMMA,DOSYM] )
     until SYM # COMMA;
     CHECKSYM( DOSYM,606 );
     STATEMENT( FSYS )
   end;

procedure CASESTAT( FSYS : SYMSET );
   begin EXPRESSION( FSYS + [OFSYM] );
     if not TESTSYM( OFSYM ) then ERROR( 607 )
     else
     repeat
        GETSYM;
        if TESTSYM( OTHERSYM ) then GETSYM
        else CONSTANTLIST( FSYS + [COLON] );
        CHECKSYM( COLON,608 );
        STATEMENT( FSYS + [ENDSYM,SEMICOLON] )
     until SYM # SEMICOLON;
     if TESTSYM( ENDSYM )
        then GETSYM
     else TEST( [],FSYS,609 )
   end;

procedure GOTOSTAT( FSYS : SYMSET );
   begin
     if TESTSYM( INTNUM )
        then GETSYM
     else ERROR( 610 )
   end;

procedure LABELSTAT( FSYS : SYMSET );
   begin
     CHECKSYM( COLON,611 );
     STATEMENT( FSYS )
   end;


procedure LOOPSTAT( FSYS : SYMSET );
   begin
     STATLIST( FSYS + [EXITSYM] );
     CHECKSYM( EXITSYM,612 );
     CHECKSYM( IFSYM,613 );
     EXPRESSION( FSYS + [SEMICOLON,ENDSYM] );
     if TESTSYM( SEMICOLON ) then
      begin GETSYM; STATLIST( FSYS + [ENDSYM] )
      end;
     CHECKSYM( ENDSYM,615 )
   end;

procedure STATEMENT;
   begin
     TEST(FSYS+[IDENT],FSYS,109);
     if TESTSYMINSet(STATBEGSYM+[IDENT]) then
      begin
        if TESTSYM(BEGINSYM) then
         begin GETSYM; STATLIST(FSYS+[ENDSYM]);
           CHECKSYM(ENDSYM,18)
         end
```

```
          if TESTSYM(IFSYM) then
           begin GETSYM; IFSTAT(FSYS)
           end
          else
             if TESTSYM(WHILESYM) then
              begin GETSYM; WHILESTAT(FSYS)
              end
             else
                if TESTSYM(REPEATSYM) then
                 begin GETSYM; REPEATSTAT(FSYS)
                 end
                else
                   if TESTSYM(IDENT) then
                    begin GETSYM;
                       OTHERSTAT(FSYS)
                    end
                   else
                   case SYM of

                      LOOPSYM  :
                          begin GETSYM; LOOPSTAT( FSYS )
                          end;
                      WITHSYM  :
                          begin GETSYM; WITHSTAT( FSYS )
                          end;
                      CASESYM  :
                          begin GETSYM; CASESTAT( FSYS )
                          end;
                      FORSYM   :
                          begin GETSYM; FORSTAT( FSYS )
                          end;
                      GOTOSYM  :
                          begin GETSYM; GOTOSTAT( FSYS )
                          end;
                      INTNUM   :
                          begin GETSYM; LABELSTAT( FSYS )
                          end
                   end
      end
   end;


procedure PARAMETIDEntlst(FSYS:SYMSET);
   begin TEST([IDENT],FSYS,121);
     if TESTSYM(IDENT) then
      begin ENTERVAR; GETSYM;
        while TESTSYM(COMMA) do
         begin GETSYM;
           if TESTSYM(IDENT) then
            begin ENTERVAR;GETSYM
            end
           else ERROR(42)
         end
      end;
     TEST(FSYS,[],122)
   end;

procedure LISTOFPARAmets(FSYS:SYMSET);
   begin
     if (TESTSYMINSet( [VARSYM,IDENT])) then
      begin
        if TESTSYM(VARSYM) then
         begin GETSYM;
           PARAMETIDEntlst(FSYS+[COLON,IDENT]);
           CHECKSYM(COLON,52);
           CHECKSYM(IDENT,53)
         end
        else
         begin
           PARAMETIDEntlst(FSYS+[COLON,IDENT]);
           CHECKSYM(COLON,54);
           CHECKSYM(IDENT,55)
         end;
```

```
          begin GETSYM;
            LISTOFPARAmets(FSYS+[VARSYM,IDENT])
          end
        end
    end;

procedure PARAMETERList(FSYS:SYMSET);
    begin
      if TESTSYM(LPAREN) then
       begin GETSYM;
         LISTOFPARAmets(FSYS+[RPAREN]);
         CHECKSYM(RPAREN,56)
       end
    end;

procedure PROCHEADER(FSYS:SYMSET);
    begin CHECKSYM(IDENT,57);
      PARAMETERList(FSYS)
    end;

procedure FUNCHEADER(FSYS:SYMSET);
    begin CHECKSYM(IDENT,58);
      PARAMETERList(FSYS+[COLON,IDENT]);
      CHECKSYM(COLON,59);
      CHECKSYM(IDENT,60);
      TEST(FSYS,[],118)
    end;

procedure BLOCK(FSYS:SYMSET);
   forward;

procedure FUNORPROCDecl(FSYS:SYMSET);
    begin
      TEST(FSYS,[],119);
      if (TESTSYMINSet( [PROCSYM,FUNCSYM])) then
       begin
         if TESTSYM(PROCSYM) then
          begin GETSYM;
            ENTERPROC;
            PROCHEADER(FSYS+[SEMICOLON])
          end
         else
          begin GETSYM;
            ENTERPROC;
            FUNCHEADER(FSYS+[SEMICOLON])
          end;
         CHECKSYM(SEMICOLON,39);
         if TESTSYMINSet([EXTERNSYM,FORWARDSYM]) then
          begin TOP:=TOP-1;GETSYM
          end
         else BLOCK(FSYS+[SEMICOLON]);
         CHECKSYM(SEMICOLON,50);
         FUNORPROCDecl(FSYS+[FUNCSYM,PROCSYM])
       end
    end;

procedure INITPROCS( FSYS : SYMSET );
    begin
      while TESTSYM( INITPROCSYm ) do
       begin
         GETSYM;
         CHECKSYM( SEMICOLON,900 );
         CHECKSYM( BEGINSYM,901 );
         repeat
            CHECKSYM( IDENT,902 );
            OTHERSTAT( FSYS + [SEMICOLON,ENDSYM] );
            if TESTSYM( SEMICOLON ) then GETSYM
         until SYM # IDENT;
         CHECKSYM( ENDSYM,903 );
         CHECKSYM( SEMICOLON,904 )
       end
    end;
```

```
procedure BLOCK;
   begin
     repeat
        if TESTSYM(CONSTSYM) then
         begin GETSYM; CONSTDECL(FSYS)
         end;
        if TESTSYM(TYPESYM) then
         begin GETSYM; TYPEDECL(FSYS)
         end;
        if TESTSYM(VARSYM) then
         begin GETSYM; VARDECL(FSYS)
         end;
        if TESTSYM( INITPROCSYm ) then
           INITPROCS( FSYS );
        if TESTSYMINSet( [PROCSYM,FUNCSYM]) then
           FUNORPROCDecl(FSYS+[BEGINSYM]);
        TEST([BEGINSYM],DECLBEGSYM+STATBEGSYM,603)
     until TESTSYMINSet(STATBEGSYM);
     CHECKSYM(BEGINSYM,40);
     STATLIST(FSYS+[ENDSYM]);
     CHECKSYM(ENDSYM,41);
     DISPLAY[TOP].CURRENT ^.DECLPROC := DISPLAY[TOP].PROCS;
     TOP := TOP - 1;
     TEST([SEMICOLON,PERIOD],FSYS,120)
   end;

procedure FILELIST(FSYS:SYMSET);
   begin TEST([IDENT],FSYS,121);
     if TESTSYM(IDENT) then
      begin GETSYM;
        while TESTSYM(COMMA) do
         begin GETSYM;
           if TESTSYM(IDENT) then GETSYM
           else ERROR(42)
         end
      end;
     TEST(FSYS,[],122)
   end;


procedure PROGRAMHEAd(FSYS:SYMSET);
   begin
     TEST([PROGSYM],FSYS,123);
     if TESTSYM(PROGSYM) then
      begin GETSYM;
        if TESTSYM(IDENT) then
         begin GETSYM;
           if TESTSYM(LPAREN) then
            begin GETSYM;
              FILELIST(FSYS+[RPAREN]);
              if TESTSYM(RPAREN) then
               begin GETSYM;
                 if TESTSYM(SEMICOLON) then GETSYM
                 else ERROR(43)
               end
              else ERROR(46)
            end
           else ERROR(45)
         end
        else ERROR(44)
      end;
     TEST(FSYS,[],124)
   end;


begin
  (* INITIALIZATIONS *)
  WORD[ 1]:='AND       ';
  WORD[ 2]:='ARRAY     ';
  WORD[ 3]:='BEGIN     ';
  WORD[ 4]:='CASE      ';
  WORD[ 5]:='CONST     ';
  WORD[ 6]:='DIV       ';
```

```
WORD[ 8]:='DOWNTO    ';
WORD[ 9]:='ELSE      ';
WORD[10]:='END       ';
WORD[11]:='EXIT      ';
WORD[12]:='EXTERN    ';
WORD[13]:='FILE      ';
WORD[14]:='FOR       ';
WORD[15]:='FORWARD   ';
WORD[16]:='FUNCTION  ';
WORD[17]:='GOTO      ';
WORD[18]:='IF        ';
WORD[19]:='IN        ';
WORD[20]:='INITPROCED';
WORD[21]:='LOOP      ';
WORD[22]:='MOD       ';
WORD[23]:='NOT       ';
WORD[24]:='OF        ';
WORD[25]:='OR        ';
WORD[26]:='OTHERS    ';
WORD[27]:='PACKED    ';
WORD[28]:='PROCEDURE ';
WORD[29]:='PROGRAM   ';
WORD[30]:='RECORD    ';
WORD[31]:='REPEAT    ';
WORD[32]:='SET       ';
WORD[33]:='THEN      ';
WORD[34]:='TO        ';
WORD[35]:='TYPE      ';
WORD[36]:='UNTIL     ';
WORD[37]:='VAR       ';
WORD[38]:='WHILE     ';
WORD[39]:='WITH      ';

WSYM[ 1]:=ANDSYM;
WSYM[ 2]:=ARRAYSYM;
WSYM[ 3]:=BEGINSYM;
WSYM[ 4]:=CASESYM;
WSYM[ 5]:=CONSTSYM;
WSYM[ 6]:=DIVSYM;
WSYM[ 7]:=DOSYM;
WSYM[ 8]:=DOWNTOSYM;
WSYM[ 9]:=ELSESYM;
WSYM[10]:=ENDSYM;
WSYM[11]:=EXITSYM;
WSYM[12]:=EXTERNSYM;
WSYM[13]:=FILESYM;
WSYM[14]:=FORSYM;
WSYM[15]:=FORWARDSYM;
WSYM[16]:=FUNCSYM;
WSYM[17]:=GOTOSYM;
WSYM[18]:=IFSYM;
WSYM[19]:=INSYM;
WSYM[20]:=INITPROCSYm;
WSYM[21]:=LOOPSYM;
WSYM[22]:=MODSYM;
WSYM[23]:=NOTSYM;
WSYM[24]:=OFSYM;
WSYM[25]:=ORSYM;
WSYM[26]:=OTHERSYM;
WSYM[27]:=PACKEDSYM;
WSYM[28]:=PROCSYM;
WSYM[29]:=PROGSYM;
WSYM[30]:=RECORDSYM;
WSYM[31]:=REPEATSYM;
WSYM[32]:=SETSYM;
WSYM[33]:=THENSYM;
WSYM[34]:=TOSYM;
WSYM[35]:=TYPESYM;
WSYM[36]:=UNTILSYM;
WSYM[37]:=VARSYM;
WSYM[38]:=WHILESYM;
WSYM[39]:=WITHSYM;
```

```
SSYM['+']:=PLUS;
SSYM['-']:=MINUS;
SSYM['*']:=TIMES;
SSYM['/']:=SLASH;
SSYM['^']:=POINTER;
SSYM['=']:=EQSYM;
SSYM['(']:=LPAREN;
SSYM[')']:=RPAREN;
SSYM['[']:=LBRACKET;
SSYM[']']:=RBRACKET;
SSYM[',']:=COMMA;
SSYM[';']:=SEMICOLON;
SSYM['#']:=NESYM;


STDPR[ 1] := 'MAINBODY. ';
STDPR[ 2] := 'READ      ';
STDPR[ 3] := 'READLN    ';
STDPR[ 4] := 'WRITE     ';
STDPR[ 5] := 'WRITELN   ';
STDPR[ 6] := 'GET       ';
STDPR[ 7] := 'PUT       ';
STDPR[ 8] := 'RESET     ';
STDPR[ 9] := 'REWRITE   ';
STDPR[10] := 'NEW       ';
STDPR[11] := 'PACK      ';
STDPR[12] := 'UNPACK    ';
STDPR[13] := 'BREAK     ';
STDPR[14] := 'PAGE      ';

CH:=' ';  CC:=0;  LL:=0;
NEWVARS := false;

DECLBEGSYM:=[CONSTSYM,VARSYM,TYPESYM,PROCSYM,FUNCSYM,
FORWARDSYM,EXTERNSYM];
STATBEGSYM:=[BEGINSYM,IFSYM,WHILESYM,REPEATSYM,FORSYM,WITHSYM,
CASESYM,GOTOSYM,INTNUM,LOOPSYM];
FACBEGSYM:=[LPAREN,NOTSYM,INTNUM,REALNUM,IDENT,STRING,
LBRACKET];
CONSTBEGSYm:=[PLUS,MINUS,INTNUM,REALNUM,STRING,IDENT];
SIMPTYBEGSym:=[STRING,LPAREN,PLUS,MINUS,IDENT,INTNUM,REALNUM];
SELECTSYS:=[POINTER,PERIOD,LBRACKET];
TYPEBEGSYM:=[PLUS,MINUS,INTNUM,REALNUM,STRING,IDENT,LPAREN,
POINTER
,PACKEDSYM,ARRAYSYM,RECORDSYM,SETSYM,FILESYM];
TYPDECL:=[RECORDSYM,ARRAYSYM,SETSYM];


GETSYM;
PROGRAMHEAd([SEMICOLON]+DECLBEGSYM+STATBEGSYM);

TOP := 0;
ENTERSTDPRocs;
TOP := 1;
DISPLAY[ TOP ]. PROCS := nil;

BLOCK([PERIOD]+STATBEGSYM+DECLBEGSYM);
CHECKSYM(PERIOD,48);
if ERRCOUNT<>0 then
 begin
    WRITELN; WRITELN;
    WRITELN (OUTPUT,ERRCOUNT,EMES);
    WRITELN (TTY,ERRCOUNT,EMES)
 end
else
 begin WRITELN (OUTPUT,NOERRMESS);
    WRITELN (TTY,NOERRMESS)
 end;
FINDCOMPS;
WRITEPINFO
end.
```

APPENDIX B

FINAL RESULT

The Strongly Connected Components of
the Call Graph, and the Called by
and Static Nesting information,
for the PASREL compiler.

```
 LIST OF PROCEDURES AND FUNCTIONS IN THE PROGRAM
 ===============================================
    with static nesting and calls information

BREAK
   It is the root of a strongly connected component,
      which consists of the following procedure(s) :
   BREAK
   It is called by the following procedures :
   MAINBODY.     READFILEID    ENDOFLINE

GET
   It is the root of a strongly connected component,
      which consists of the following procedure(s) :
   GET
   It is called by the following procedures :
   NEXTCH

MAINBODY.
   It is the root of a strongly connected component,
      which consists of the following procedure(s) :
   MAINBODY.
   It calls the following procedures :
   WRITE         ENDOFLINE     BLOCK         INSYMBOL      GETNEXTLIN
   BREAK         WRITELN       REWRITE       READFILEID    ENTERDEBNA
   ENTERUNDEC    ENTERSTDNA    ENTERSTDTY

NEW
   It is the root of a strongly connected component,
      which consists of the following procedure(s) :
   NEW
   It is called by the following procedures :
   ENTERDEBNA    ENTERUNDEC    ENTERSTDNA    ENTERSTDTY    BODY
   CASESTATEM    GETNEWGLOB    FACTOR        GETSTRINGA    DEPCST
   PROCEDURED    PARAMETERL    VARIABLEDE    TYPEDECLAR    CONSTANTDE
   LABELDECLA    TYP           FIELDLIST     RECSECTION    SIMPLETYPE
   COMPTYPES     CONSTANT      INSYMBOL      ERRORWITHT

PACK          is not called in this program.

PAGE
   It is the root of a strongly connected component,
      which consists of the following procedure(s) :
   PAGE
   It is called by the following procedures :
   GETNEXTLIN

PUT
   It is the root of a strongly connected component,
      which consists of the following procedure(s) :
   PUT
   It is called by the following procedures :
   PUTRELCODE

READ
   It is the root of a strongly connected component,
      which consists of the following procedure(s) :
   READ
   It is called by the following procedures :
   NEXTCH

READLN
   It is the root of a strongly connected component,
      which consists of the following procedure(s) :
   READLN
   It is called by the following procedures :
   READFILEID    ENDOFLINE     GETNEXTLIN

RESET
   It is the root of a strongly connected component,
      which consists of the following procedure(s) :
   RESET
   It is called by the following procedures :
   READFILEID
```

```
REWRITE
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    REWRITE
    It is called  by the following procedures :
    MAINBODY.

UNPACK
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    UNPACK
    It is called  by the following procedures :
    MCCODE

WRITELN
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    WRITELN
    It is called  by the following procedures :
    MAINBODY.      READFILEID   WRITEMC      WRITEHEADE   WRITEFIRST
    WRITEIDENT     NEUEZEILE    INSYMBOL     ENDOFLINE    GETNEXTLIN
    WRITEBUFFE

WRITE
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    WRITE
    It is called  by the following procedures :
    MAINBODY.      READFILEID   MCCODE       WRITEHEADE   WRITEFIRST
    WRITEIDENT     WRITEWORD    SHOWRELOCA   NEUEZEILE    ENDOFLINE
    GETNEXTLIN

   Following procedures are declared in MAINBODY.
   -------------------------------------------------

BLOCK
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    BLOCK          PROCEDURED
    It calls the  following procedures :
    SKIPIFERR      ERRORWITHT   PROCEDURED   BODY         ERRANDSKIP
    ERROR          VARIABLEDE   TYPEDECLAR   CONSTANTDE   LABELDECLA
    INSYMBOL
    It is called  by the following procedures :
    MAINBODY.      PROCEDURED

ENDOFLINE
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    ENDOFLINE
    It calls the  following procedures :
    GETNEXTLIN     READLN       BREAK        WRITE        WRITELN

    It is called  by the following procedures :
    MAINBODY.      INSYMBOL     OPTIONS

ENTERDEBNA
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    ENTERDEBNA
    It calls the  following procedures :
    ENTERID        NEW
    It is called  by the following procedures :
    MAINBODY.

ENTERID
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    ENTERID
    It calls the  following procedures :
    ERROR
    It is called  by the following procedures :
    ENTERDEBNA     ENTERSTDNA   PROCEDURED   PARAMETERL
    TYPEDECLAR     CONSTANTDE   FIELDLIST    SIMPLETYPE
```

```
ENTERSTDNA
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    ENTERSTDNA
    It calls the following procedures :
    ENTERID         NEW
    It is called by the following procedures :
    MAINBODY.

ENTERSTDTY
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    ENTERSTDTY
    It calls the following procedures :
    NEW
    It is called by the following procedures :
    MAINBODY.

ENTERUNDEC
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    ENTERUNDEC
    It calls the following procedures :
    NEW
    It is called by the following procedures :
    MAINBODY.

ERRORWITHT
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    ERRORWITHT
    It calls the following procedures :
    NEW             ERROR
    It is called by the following procedures :
    BLOCK           STOREWORD       MACRO           FULLWORD        VARIABLEDE
    TYPEDECLAR

ERROR
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    ERROR
    It is called by the following procedures :
    BLOCK           BODY            STATEMENT       WITHSTATEM      LOOPSTATEM
    FORSTATEME      WHILESTATE      REPEATSTAT      CASESTATEM      IFSTATEMEN
    COMPOUNDST      GOTOSTATEM      ASSIGNMENT      STOREGLOBA      EXPRESSION
    SIMPLEEXPR      TERM            FACTOR          CALL            CALLNONSTA
    PROTECTION      EOFEOLN         PREDSUCC        CHR             ORD
    ODD             TRUNC           SQR             ABS             GETLINENR
    RELEASE         MARK            NEW             UNPACK          PACK
    WRITEWRITE      READREADLN      GETSTRINGA      VARIABLE        GETFILENAM
    SELECTOR        LOADADDRES      MACRO           INCREMENTR      PROCEDURED
    PARAMETERL      VARIABLEDE      TYPEDECLAR      CONSTANTDE      LABELDECLA
    TYP             FIELDLIST       SIMPLETYPE      CONSTANT        SKIPIFERR
    SEARCHID        ENTERID         INSYMBOL        OPTIONS         ERRORWITHT


GETBOUNDS
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    GETBOUNDS
    It is called by the following procedures :
    ASSIGNMENT      NEW             UNPACK          PACK            WRITEWRITE
    SELECTOR        TYP             COMPTYPES

GETNEXTLIN
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    GETNEXTLIN
    It calls the following procedures :
    WRITE           READLN          WRITELN         PAGE            NEWPAGER

    It is called by the following procedures :
    MAINBODY.       ENDOFLINE
```

```
INSYMBOL
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    INSYMBOL
    It calls the following procedures :
    INSYMBOL    NEW         ERROR       WRITELN     WRITEBUFFE
    OPTIONS     NEXTCH      ENDOFLINE
    It is called by the following procedures :
    MAINBODY.   BLOCK       BODY        STATEMENT   WITHSTATEM
    LOOPSTATEM  FORSTATEME  WHILESTATE  REPEATSTAT  CASESTATEM
    IFSTATEMEN  COMPOUNDST  GOTOSTATEM  ASSIGNMENT  EXPRESSION
    SIMPLEEXPR  TERM        FACTOR      CALL        CALLNONSTA
    NEW         UNPACK      PACK        WRITEWRITE  READREADLN
    GETPUTRESE  GETSTRINGA  VARIABLE    GETFILENAM  SELECTOR
    PROCEDURED  PARAMETERL  VARIABLEDE  TYPEDECLAR  CONSTANTDE
    LABELDECLA  TYP         FIELDLIST   SIMPLETYPE  CONSTANT
    SKIPIFERR   INSYMBOL

NEWPAGER
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    NEWPAGER
    It is called by the following procedures :
    MCCODE      GETNEXTLIN

READFILEID
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    READFILEID
    It calls the following procedures :
    RESET       READLN      BREAK       WRITE       WRITELN
    OPERAND
    It is called by the following procedures :
    MAINBODY.

SEARCHID
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    SEARCHID
    It calls the following procedures :
    ERROR
    It is called by the following procedures :
    STATEMENT   WITHSTATEM  FORSTATEME  FACTOR      CALLNONSTA
    GETINTEGER  VARIABLE    GETFILENAM  PROCEDURED  PARAMETERL
    TYP         FIELDLIST   SIMPLETYPE  CONSTANT

SEARCHSECT
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    SEARCHSECT
    It is called by the following procedures :
    SELECTOR    PROCEDURED

WRITEBUFFE
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    WRITEBUFFE
    It calls the following procedures :
    WRITELN
    It is called by the following procedures :
    MCCODE      MCGLOBALS   INSYMBOL


  Following procedures are declared in READFILEID
  -----------------------------------------------

OPERAND
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    OPERAND
    It calls the following procedures :
    SETSTATUS   READCHAR    READOCTAL   NEXTCH
    It is called by the following procedures :
    READFILEID
```

```
  Following procedures are declared in OPERAND
  ================================================

NEXTCH
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    NEXTCH
    It calls the following procedures :
    READ
    It is called by the following procedures :
    OPERAND

READCHAR
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    READCHAR
    It is called by the following procedures :
    OPERAND

READOCTAL
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    READOCTAL
    It is called by the following procedures :
    OPERAND

SETSTATUS
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    SETSTATUS
    It is called by the following procedures :
    OPERAND

  Following procedures are declared in INSYMBOL
  ================================================

NEXTCH
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    NEXTCH
    It calls the following procedures :
    GET
    It is called by the following procedures :
    INSYMBOL     OPTIONS

OPTIONS
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    OPTIONS
    It calls the following procedures :
    ENDOFLINE    ERROR        NEXTCH
    It is called by the following procedures :
    INSYMBOL

  Following procedures are declared in BLOCK
  ================================================

BODY
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    BODY
    It calls the following procedures :
    INSERTADDR   LEAVEBODY    NEW          ENTERBODY    ERROR
    INSYMBOL     STATEMENT    WRITEMC
    It is called by the following procedures :
    BLOCK
```

```
COMPTYPES
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    COMPTYPES
    It calls the following procedures :
    GETBOUNDS     NEW           COMPTYPES
    It is called by the following procedures :
    FORSTATEME    CASESTATEM    ASSIGNMENT    EXPRESSION    SIMPLEEXPR
    TERM          FACTOR        CALLNONSTA    GETLINENR     MARK
    NEW           UNPACK        PACK          WRITEWRITE    READREADLN
    GETPUTRESE    GETSTRINGA    GETFILENAM    SELECTOR      MCFILEBLOC
    FIELDLIST     STRING        COMPTYPES

CONSTANT
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    CONSTANT
    It calls the following procedures :
    IFERRSKIP     ERRANDSKIP    ERROR         SEARCHID      INSYMBOL
    NEW           SKIPIFERR
    It is called by the following procedures :
    CASESTATEM    FACTOR        NEW           CONSTANTDE    FIELDLIST
    SIMPLETYPE

CONSTANTDE
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    CONSTANTDE
    It calls the following procedures :
    IFERRSKIP     ENTERID       CONSTANT      ERROR         INSYMBOL
    NEW           ERRANDSKIP    SKIPIFERR
    It is called by the following procedures :
    BLOCK

ERRANDSKIP
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    ERRANDSKIP
    It calls the following procedures :
    SKIPIFERR
    It is called by the following procedures :
    BLOCK         STATEMENT     LOOPSTATEM    FORSTATEME    FACTOR
    CALLNONSTA    GETPUTRESE    PROCEDURED    PARAMETERL    VARIABLEDE
    TYPEDECLAR    CONSTANTDE    FIELDLIST     CONSTANT

IFERRSKIP
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    IFERRSKIP
    It calls the following procedures :
    SKIPIFERR
    It is called by the following procedures :
    FACTOR        CALLNONSTA    SELECTOR      PROCEDURED    PARAMETERL
    VARIABLEDE    TYPEDECLAR    CONSTANTDE    LABELDECLA    TYP
    FIELDLIST     SIMPLETYPE    CONSTANT

LABELDECLA
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    LABELDECLA
    It calls the following procedures :
    IFERRSKIP     NEW           ERROR         INSYMBOL
    It is called by the following procedures :
    BLOCK

PROCEDURED
    It calls the following procedures :
    SKIPIFERR     BLOCK         IFERRSKIP     ERRANDSKIP    SEARCHID
    PARAMETERL    INSYMBOL      ENTERID       NEW           ERROR
    SEARCHSECT
    It is called by the following procedures :
    BLOCK
```

```
SKIPIFERR
     It is the root of a strongly connected component,
         which consists of the following procedure(s) :
     SKIPIFERR
     It calls the following procedures :
     INSYMBOL        ERROR
     It is called by the following procedures :
     BLOCK           STATEMENT       PROCEDURED      PARAMETERL      VARIABLEDE
     TYPEDECLAR      CONSTANTDE      TYP             FIELDLIST       SIMPLETYPE
     CONSTANT        ERRANDSKIP      IFERRSKIP

STRING
     It is the root of a strongly connected component,
         which consists of the following procedure(s) :
     STRING
     It calls the following procedures :
     COMPTYPES
     It is called by the following procedures :
     EXPRESSION      NEW             WRITEWRITE      LOADADDRES      FIELDLIST
     SIMPLETYPE

TYP
     It is the root of a strongly connected component,
         which consists of the following procedure(s) :
     TYP             FIELDLIST
     It calls the following procedures :
     IFERRSKIP       FIELDLIST       GETBOUNDS       TYP             ERROR
     SEARCHID        INSYMBOL        NEW             SIMPLETYPE      SKIPIFERR

     It is called by the following procedures :
     VARIABLEDE      TYPEDECLAR      TYP             FIELDLIST

TYPEDECLAR
     It is the root of a strongly connected component,
         which consists of the following procedure(s) :
     TYPEDECLAR
     It calls the following procedures :
     ERRORWITHT      IFERRSKIP       ENTERID         TYP             ERROR
     INSYMBOL        NEW             ERRANDSKIP      SKIPIFERR
     It is called by the following procedures :
     BLOCK

VARIABLEDE
     It is the root of a strongly connected component,
         which consists of the following procedure(s) :
     VARIABLEDE
     It calls the following procedures :
     ERRORWITHT      IFERRSKIP       TYP             SKIPIFERR       ERROR
     INSYMBOL        ENTERID         NEW             ERRANDSKIP
     It is called by the following procedures :
     BLOCK


   Following procedures are declared in TYP
   ---------------------------------------------

FIELDLIST
     It calls the following procedures :
     IFERRSKIP       FIELDLIST       CONSTANT        COMPTYPES       STRING
     SEARCHID        ERRANDSKIP      RECSECTION      TYP             ERROR
     INSYMBOL        ENTERID         NEW             SKIPIFERR
     It is called by the following procedures :
     TYP             FIELDLIST

LOG2
     It is the root of a strongly connected component,
         which consists of the following procedure(s) :
     LOG2
     It is called by the following procedures :
     SIMPLETYPE
```

```
FULLWORD
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    FULLWORD
    It calls the following procedures :
    ERRORWITHT
    It is called by the following procedures :
    CASESTATEM   ENTERBODY    PUTPAGER

GETPARADDR
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    GETPARADDR
    It calls the following procedures :
    MACRO5       INCREMENTR   FETCHBASIS
    It is called by the following procedures :
    WITHSTATEM   SELECTOR

INCREMENTR
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    INCREMENTR
    It calls the following procedures :
    ERROR
    It is called by the following procedures :
    ASSIGNMENT   EXPRESSION   FACTOR       TIME         RUNTIME
    NEW          UNPACK       PACK         WRITEWRITE   GETPUTRESE
    SELECTOR     LOADADDRES   LOAD         MAKECODE     GETPARADDR


INSERTADDR
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    INSERTADDR
    It is called by the following procedures :
    BODY         STATEMENT    LOOPSTATEM   FORSTATEME   WHILESTATE
    REPEATSTAT   CASESTATEM   INSERTBOUN   IFSTATEMEN   GOTOSTATEM
    LEAVEBODY    DEPCST

LEAVEBODY
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    LEAVEBODY
    It calls the following procedures :
    INSERTADDR   MACRO3R      SUPPORT      MACRO3       MACRO4
    PUTLINER
    It is called by the following procedures :
    BODY

LOAD
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    LOAD
    It calls the following procedures :
    MAKECODE     INCREMENTR
    It is called by the following procedures :
    FORSTATEME   CASESTATEM   ASSIGNMENT   EXPRESSION   SIMPLEEXPR
    TERM         FACTOR       SEARCHCODE   CALL         CALLNONSTA
    PROTECTION   PUT8BITSTO   RELEASE      NEW          UNPACK
    PACK         WRITEWRITE   GETPUTRESE   SELECTOR     MAKEREAL


LOADADDRES
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    LOADADDRES
    It calls the following procedures :
    MACRO        FETCHBASIS   ERROR        DEPCST       MACRO3
    STRING       INCREMENTR
    It is called by the following procedures :
    ASSIGNMENT   EXPRESSION   CALLNONSTA   EOFEOLN      GETINTEGER
    MARK         UNPACK       PACK         WRITEWRITE   READREADLN
    GETPUTRESE   GETSTRINGA   GETFILENAM
```

```
MACRO
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    MACRO
    It calls the following procedures :
    ERROR           ERRORWITHT
    It is called by the following procedures :
    FORSTATEME      CASESTATEM      ASSIGNMENT      LOADADDRES      STORE
    MAKECODE        MACRO3R         MACRO4R         MACRO3          MACRO4
    MACRO5

MACRO3
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    MACRO3
    It calls the following procedures :
    MACRO
    It is called by the following procedures :
    WITHSTATEM      FORSTATEME      CASESTATEM      IFSTATEMEN      GOTOSTATEM
    ASSIGNMENT      EXPRESSION      SIMPLEEXPR      FACTOR          CALLNONSTA
    PROTECTION      EOFEOLN         PREDSUCC        ODD             TRUNC
    SQR             TIME            ABS             RUNTIME         PUT8BITSTO
    RELEASE         NEW             UNPACK          PACK            WRITEWRITE
    GETPUTRESE      SELECTOR        SUBLOWBOUN      MAKEREAL        LOADADDRES
    MAKECODE        FETCHBASIS      LEAVEBODY       ENTERBODY       PUTLINER

MACRO3R
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    MACRO3R
    It calls the following procedures :
    MACRO
    It is called by the following procedures :
    LOOPSTATEM      FORSTATEME      WHILESTATE      GOTOSTATEM      EXPRESSION
    CALLNONSTA      PREDSUCC        NEW             UNPACK          PACK
    SELECTOR        STORE           MAKECODE        LEAVEBODY       ENTERBODY
    SUPPORT         PUTLINER

MACRO4
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    MACRO4
    It calls the following procedures :
    MACRO
    It is called by the following procedures :
    FORSTATEME      ASSIGNMENT      EXPRESSION      FACTOR          CALLNONSTA
    EOFEOLN         GETLINENR       MARK            NEW             UNPACK
    PACK            SELECTOR        FETCHBASIS      LEAVEBODY       ENTERBODY

MACRO4R
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    MACRO4R
    It calls the following procedures :
    MACRO
    It is called by the following procedures :
    EXPRESSION      PUTLINER        PUTPAGER

MACRO5
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    MACRO5
    It calls the following procedures :
    MACRO
    It is called by the following procedures :
    ASSIGNMENT      STORE           MAKECODE        GETPARADDR

MAKECODE
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    MAKECODE
    It calls the following procedures :
    MACRO3R         MACRO           MACRO5          INCREMENTR      FETCHBASIS
    DEPCST          MACRO3
    It is called by the following procedures :
    FORSTATEME      ASSIGNMENT      SEARCHCODE      PREDSUCC        LOAD
```

```
PUTLINER
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    PUTLINER
    It calls the following procedures :
    MACRO4R       MACRO3        MACRO3R       PUTPAGER
    It is called by the following procedures :
    STATEMENT     LEAVEBODY

PUTPAGER
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    PUTPAGER
    It calls the following procedures :
    FULLWORD      MACRO4R
    It is called by the following procedures :
    PUTLINER

STATEMENT
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    STATEMENT     WITHSTATEM    FORSTATEME    LOOPSTATEM    REPEATSTAT
    WHILESTATE    CASESTATEM    IFSTATEMEN    COMPOUNDST
    It calls the following procedures :
    SKIPIFERR     WITHSTATEM    FORSTATEME    LOOPSTATEM    REPEATSTAT
    WHILESTATE    CASESTATEM    IFSTATEMEN    GOTOSTATEM    COMPOUNDST
    ASSIGNMENT    CALL          SEARCHID      ERRANDSKIP    PUTLINER
    INSYMBOL      INSERTADDR    ERROR
    It is called by the following procedures :
    BODY          WITHSTATEM    LOOPSTATEM    FORSTATEME    WHILESTATE
    REPEATSTAT    CASESTATEM    IFSTATEMEN    COMPOUNDST

STORE
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    STORE
    It calls the following procedures :
    MACRO3R       MACRO         MACRO5        FETCHBASIS
    It is called by the following procedures :
    ASSIGNMENT    GETLINENR     NEW

SUPPORT
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    SUPPORT
    It calls the following procedures :
    MACRO3R
    It is called by the following procedures :
    ASSIGNMENT    PREDSUCC      TRUNC         PAGE          NEW
    UNPACK        PACK          WRITEWRITE    BREAK         READREADLN
    GETPUTRESE    SUBLOWBOUN    MAKEREAL      LEAVEBODY     ENTERBODY


WRITEMC
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    WRITEMC
    It calls the following procedures :
    WRITELN       MCLIBRARY     MCVARIOUS     MCSYMBOLS     MCCODE
    MCGLOBALS     MCFILEBLOC
    It is called by the following procedures :
    BODY

  Following procedures are declared in WRITEMC.
  ---------------------------------------------

MCCODE
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    MCCODE
    It calls the following procedures :
    WRITEPAIR     WRITERECOR    NEWPAGER      COPYCTP       UNPACK
    SHOWRELOCA    WRITEWORD     WRITE         NEUEZEILE     WRITEBLOCK
    WRITEFIRST    WRITEBUFFE
    It is called by the following procedures :
    WRITEMC
```

```
MCFILEBLOC
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    MCFILEBLOC
    It calls the following procedures :
    COMPTYPES      WRITEWORD      WRITEBLOCK     WRITEFIRST
    It is called by the following procedures :
    WRITEMC

MCGLOBALS
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    MCGLOBALS
    It calls the following procedures :
    WRITEWORD      WRITEBLOCK     WRITEFIRST     WRITEBUFFE
    It is called by the following procedures :
    WRITEMC

MCLIBRARY
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    MCLIBRARY
    It calls the following procedures :
    WRITEPAIR      WRITEIDENT     WRITEBLOCK     WRITEHEADE
    It is called by the following procedures :
    WRITEMC

MCSYMBOLS
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    MCSYMBOLS
    It calls the following procedures :
    WRITEPAIR      WRITEIDENT     WRITEBLOCK     WRITEHEADE
    It is called by the following procedures :
    WRITEMC

MCVARIOUS
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    MCVARIOUS
    It calls the following procedures :
    WRITEIDENT     PUTRELCODE     WRITEPAIR      WRITEBLOCK     WRITEHEADE

    It is called by the following procedures :
    WRITEMC

NEUEZEILE
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    NEUEZEILE
    It calls the following procedures :
    WRITE          WRITELN
    It is called by the following procedures :
    MCCODE         WRITEWORD

PUTRELCODE
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    PUTRELCODE
    It calls the following procedures :
    PUT
    It is called by the following procedures :
    MCVARIOUS      WRITEWORD      WRITEBLOCK

RADIX50
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    RADIX50
    It is called by the following procedures :
    WRITEIDENT
```

```
SHOWRELOCA
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    SHOWRELOCA
    It calls the following procedures :
    WRITE
    It is called by the following procedures :
    MCCODE          WRITEWORD

WRITEBLOCK
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    WRITEBLOCK
    It calls the following procedures :
    PUTRELCODE
    It is called by the following procedures :
    MCLIBRARY       MCSYMBOLS       MCVARIOUS       MCCODE          MCGLOBALS
    MCFILEBLOC      WRITEWORD

WRITEFIRST
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    WRITEFIRST
    It calls the following procedures :
    WRITE           WRITELN
    It is called by the following procedures :
    MCCODE          MCGLOBALS       MCFILEBLOC

WRITEHEADE
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    WRITEHEADE
    It calls the following procedures :
    WRITE           WRITELN
    It is called by the following procedures :
    MCLIBRARY       MCSYMBOLS       MCVARIOUS

WRITEIDENT
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    WRITEIDENT
    It calls the following procedures :
    WRITEWORD       RADIX50         WRITE           WRITELN
    It is called by the following procedures :
    MCLIBRARY       MCSYMBOLS       MCVARIOUS
```

```
WRITEPAIR
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    WRITEPAIR
    It calls the following procedures :
    WRITEWORD
    It is called by the following procedures :
    MCLIBRARY    MCSYMBOLS    MCVARIOUS    MCCODE

WRITEWORD
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    WRITEWORD
    It calls the following procedures :
    SHOWRELOCA   WRITE        NEUEZEILE    PUTRELCODE   WRITEBLOCK

    It is called by the following procedures :
    MCCODE       WRITERECOR   MCGLOBALS    MCFILEBLOC   WRITEIDENT
    WRITEPAIR


     Following procedures are declared in MCCODE
     -------------------------------------------

CONSTRECSI
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    CONSTRECSI
    It is called by the following procedures :
    COPYCSP

COPYCSP
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    COPYCSP
    It calls the following procedures :
    WRITERECOR   CONSTRECSI
    It is called by the following procedures :
    COPYCTP

COPYCTP
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    COPYCTP      COPYSTP
    It calls the following procedures :
    COPYCSP      COPYSTP      COPYCTP      WRITERECOR
    It is called by the following procedures :
    MCCODE       COPYSTP      COPYCTP

COPYSTP
    It calls the following procedures :
    COPYSTP      COPYCTP      WRITERECOR
    It is called by the following procedures :
    COPYSTP      COPYCTP

WRITERECOR
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    WRITERECOR
    It calls the following procedures :
    WRITEWORD
    It is called by the following procedures :
    MCCODE       COPYSTP      COPYCTP      COPYCSP


     Following procedures are declared in STATEMENT
     ----------------------------------------------

ASSIGNMENT
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    ASSIGNMENT
    It calls the following procedures :
    MACRO4       MACRO3       INCREMENTR   MACRO5       LOADADDRES
    SUPPORT      MAKECODE     GETBOUNDS    STORE        MAKEREAL
    LOAD         MACRO        FETCHBASIS   ERROR        STOREGLOBA
    COMPTYPES    EXPRESSION   INSYMBOL     SELECTOR
    It is called by the following procedures :
    STATEMENT
```

```
CALL
    It calls the following procedures :
    CALLNONSTA    EOFEOLN       PREDSUCC      CHR           ORD
    ODD           TRUNC         SQR           ABS           TIME
    RUNTIME       LOAD          EXPRESSION    PROTECTION    PAGE
    PUT8BITSTO    GETLINENR     RELEASE       MARK          NEW
    UNPACK        PACK          WRITEWRITE    BREAK         READREADLN
    GETPUTRESE    ERROR         INSYMBOL
    It is called by the following procedures :
    STATEMENT     FACTOR

CASESTATEM
    It calls the following procedures :
    FULLWORD      INSERTADDR    INSERTBOUN    STATEMENT     NEW
    COMPTYPES     CONSTANT      INSYMBOL      ERROR         MACRO
    MACRO3        LOAD          EXPRESSION
    It is called by the following procedures :
    STATEMENT

COMPOUNDST
    It calls the following procedures :
    ERROR         INSYMBOL      STATEMENT
    It is called by the following procedures :
    STATEMENT

EXPRESSION
    It calls the following procedures :
    MACRO4R       MACRO3R       CHANGEBOOL    SEARCHCODE    STRING
    MAKEREAL      ERROR         MACRO4        LOAD          COMPTYPES
    INSYMBOL      LOADADDRES    MACRO3        INCREMENTR    SIMPLEEXPR

    It is called by the following procedures :
    LOOPSTATEM    FORSTATEME    WHILESTATE    REPEATSTAT    CASESTATEM
    IFSTATEMEN    ASSIGNMENT    FACTOR        CALL          CALLNONSTA
    PROTECTION    PUT8BITSTO    RELEASE       NEW           UNPACK
    PACK          WRITEWRITE    GETPUTRESE    GETSTRINGA    SELECTOR


FORSTATEME
    It calls the following procedures :
    INSERTADDR    MACRO3R       STATEMENT     MACRO3        MAKECODE
    MACRO         FETCHBASIS    MACRO4        LOAD          EXPRESSION
    ERRANDSKIP    INSYMBOL      COMPTYPES     ERROR         SEARCHID

    It is called by the following procedures :
    STATEMENT

GOTOSTATEM
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    GOTOSTATEM
    It calls the following procedures :
    INSERTADDR    MACRO3        MACRO3R       INSYMBOL      ERROR

    It is called by the following procedures :
    STATEMENT

IFSTATEMEN
    It calls the following procedures :
    INSERTADDR    MACRO3        STATEMENT     ERROR         INSYMBOL
    EXPRESSION
    It is called by the following procedures :
    STATEMENT

LOOPSTATEM
    It calls the following procedures :
    ERROR         INSERTADDR    MACRO3R       ERRANDSKIP    EXPRESSION
    INSYMBOL      STATEMENT
    It is called by the following procedures :
    STATEMENT

MAKEREAL
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    MAKEREAL
    It calls the following procedures :
    MAKEREAL      SUPPORT       MACRO3        LOAD
    It is called by the following procedures :
    ASSIGNMENT    EXPRESSION    SIMPLEEXPR    TERM          CALLNONSTA
    MAKEREAL
```

```
REPEATSTAT
    It calls the following procedures :
    ERROR          INSERTADDR     EXPRESSION     INSYMBOL       STATEMENT

    It is called by the following procedures :
    STATEMENT

SELECTOR
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    SELECTOR       EXPRESSION     SIMPLEEXPR     TERM           FACTOR
    CALL           CALLNONSTA     PROTECTION     PAGE           GETFILENAM
    PUT8BITSTO     GETLINENR      VARIABLE       RELEASE        MARK
    NEW            UNPACK         PACK           WRITEWRITE     BREAK
    READREADLN     GETPUTRESE     GETSTRINGA
    It calls the following procedures :
    SEARCHSECT     MACRO3R        MACRO4         INCREMENTR     MACRO3
    SUBLOWBOUN     GETBOUNDS      COMPTYPES      LOAD           EXPRESSION
    INSYMBOL       GETPARADDR     IFERRSKIP      ERROR
    It is called by the following procedures :
    WITHSTATEM     ASSIGMENT      FACTOR         GETINTEGER     VARIABLE
    GETFILENAM

WHILESTATE
    It calls the following procedures :
    INSERTADDR     MACRO3R        STATEMENT      ERROR          INSYMBOL
    EXPRESSION
    It is called by the following procedures :
    STATEMENT

WITHSTATEM
    It calls the following procedures :
    STATEMENT      MACRO3         FETCHBASIS     GETPARADDR     SELECTOR
    ERROR          INSYMBOL       SEARCHID
    It is called by the following procedures :
    STATEMENT


  Following procedures are declared in SELECTOR
  ===============================================

SUBLOWBOUN
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    SUBLOWBOUN
    It calls the following procedures :
    SUPPORT        MACRO3
    It is called by the following procedures :
    SELECTOR


  Following procedures are declared in EXPRESSION
  ===============================================

CHANGEBOOL
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    CHANGEBOOL
    It is called by the following procedures :
    EXPRESSION

SEARCHCODE
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    SEARCHCODE
    It calls the following procedures :
    LOAD           CHANGEOPER     MAKECODE
    It is called by the following procedures :
    EXPRESSION     SIMPLEEXPR     TERM

SIMPLEEXPR
    It calls the following procedures :
    MAKEREAL       COMPTYPES      SEARCHCODE     ERROR          MACRO3
    LOAD           TERM           INSYMBOL
    It is called by the following procedures :
    EXPRESSION
```

```
   Following procedures are declared in SIMPLEEXPR
   ------------------------------------------------

TERM
    It calls the following procedures :
    ERROR         MAKEREAL      COMPTYPES     SEARCHCODE    INSYMBOL
    LOAD          FACTOR
    It is called by the following procedures :
    SIMPLEEXPR

   Following procedures are declared in TERM
   ------------------------------------------------

FACTOR
    It calls the following procedures :
    IFERRSKIP     DEPCST        MACRO4        COMPTYPES     INCREMENTR
    NEW           MACRO3        FACTOR        ERROR         EXPRESSION
    CONSTANT      SELECTOR      LOAD          CALL          INSYMBOL
    SEARCHID      ERRANDSKIP
    It is called by the following procedures :
    TERM          FACTOR

   Following procedures are declared in SEARCHCODE
   ------------------------------------------------

CHANGEOPER
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    CHANGEOPER
    It is called by the following procedures :
    SEARCHCODE

   Following procedures are declared in CASESTATEM
   ------------------------------------------------

INSERTBOUN
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    INSERTBOUN
    It calls the following procedures :
    DEPCST        INSERTADDR
    It is called by the following procedures :
    CASESTATEM

   Following procedures are declared in CALL
   ------------------------------------------------

ABS
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    ABS
    It calls the following procedures :
    ERROR         MACRO3
    It is called by the following procedures :
    CALL

BREAK
    It calls the following procedures :
    SUPPORT       GETFILENAM
    It is called by the following procedures :
    CALL

CALLNONSTA
    It calls the following procedures :
    MACRO3R       LOADADDRES    MAKEREAL      LOAD          EXPRESSION
    IFERRSKIP     COMPTYPES     SEARCHID      ERRANDSKIP    INSYMBOL
    ERROR         MACRO4        MACRO3
    It is called by the following procedures :
    CALL
```

```
CHR
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    CHR
    It calls the following procedures :
    ERROR
    It is called by the following procedures :
    CALL

EOFEOLN
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    EOFEOLN
    It calls the following procedures :
    MACRO3          MACRO4          LOADADDRES      ERROR
    It is called by the following procedures :
    CALL

GETFILENAM
    It calls the following procedures :
    LOADADDRES      SELECTOR        ERROR           COMPTYPES       SEARCHID
    INSYMBOL
    It is called by the following procedures :
    PAGE            GETLINENR       WRITEWRITE      BREAK           READREADLN


GETINTEGER
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    GETINTEGER
    It calls the following procedures :
    LOADADDRES      SELECTOR        SEARCHID

GETLINENR
    It calls the following procedures :
    ERROR           STORE           MACRO4          COMPTYPES       VARIABLE
    GETFILENAM
    It is called by the following procedures :
    CALL

GETPUTRESE
    It calls the following procedures :
    SUPPORT         COMPTYPES       MACRO3          INCREMENTR      LOAD
    EXPRESSION      INSYMBOL        GETSTRINGA      ERRANDSKIP      LOADADDRES
    VARIABLE
    It is called by the following procedures :
    CALL

MARK
    It calls the following procedures :
    ERROR           MACRO4          LOADADDRES      COMPTYPES       VARIABLE

    It is called by the following procedures :
    CALL

NEW
    It calls the following procedures :
    STORE           MACRO3R         SUPPORT         MACRO4          MACRO3
    LOAD            GETBOUNDS       EXPRESSION      COMPTYPES       STRING
    CONSTANT        INSYMBOL        ERROR           VARIABLE        INCREMENTR

    It is called by the following procedures :
    CALL

ODD
    It is the root of a strongly connected component,
       which consists of the following procedure(s) :
    ODD
    It calls the following procedures :
    MACRO3          ERROR
    It is called by the following procedures :
    CALL
```

```
ORD
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    ORD
    It calls the following procedures :
    ERROR
    It is called by the following procedures :
    CALL

PACK
    It calls the following procedures :
    MACRO3R         INCREMENTR   MACRO4        SUPPORT       LOAD
    MACRO3          GETBOUNDS    COMPTYPES     EXPRESSION    INSYMBOL
    ERROR           LOADADDRES   VARIABLE
    It is called by the following procedures :
    CALL

PAGE
    It calls the following procedures :
    SUPPORT         GETFILENAM
    It is called by the following procedures :
    CALL

PREDSUCC
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    PREDSUCC
    It calls the following procedures :
    MAKECODE        SUPPORT      MACRO3        MACRO3R       ERROR

    It is called by the following procedures :
    CALL

PROTECTION
    It calls the following procedures :
    ERROR           MACRO3       LOAD          EXPRESSION
    It is called by the following procedures :
    CALL

PUTBBITSTO
    It calls the following procedures :
    MACRO3          LOAD         EXPRESSION
    It is called by the following procedures :
    CALL

READREADLN
    It calls the following procedures :
    INSYMBOL        SUPPORT      ERROR         COMPTYPES     LOADADDRES
    VARIABLE        GETFILENAM
    It is called by the following procedures :
    CALL

RELEASE
    It calls the following procedures :
    ERROR           MACRO3       LOAD          EXPRESSION
    It is called by the following procedures :
    CALL

RUNTIME
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    RUNTIME
    It calls the following procedures :
    MACRO3          INCREMENTR
    It is called by the following procedures :
    CALL

SQR
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    SQR
    It calls the following procedures :
    ERROR           MACRO3
    It is called by the following procedures :
    CALL

TIME
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    TIME
    It calls the following procedures :
    MACRO3          INCREMENTR
    It is called by the following procedures :
    CALL
```

```
TRUNC
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    TRUNC
    It calls the following procedures :
    SUPPORT         MACRO3          ERROR
    It is called by the following procedures :
    CALL

UNPACK
    It calls the following procedures :
    MACRO3R         MACRO4          SUPPORT         MACRO3          LOAD
    GETBOUNDS       INCREMENTR      COMPTYPES       EXPRESSION      INSYMBOL
    ERROR           LOADADDRES      VARIABLE
    It is called by the following procedures :
    CALL

VARIABLE
    It calls the following procedures :
    SELECTOR        ERROR           INSYMBOL        SEARCHID
    It is called by the following procedures :
    GETLINENR       MARK            NEW             UNPACK          PACK
    READREADLN      GETPUTRESE

WRITEWRITE
    It calls the following procedures :
    SUPPORT         GETBOUNDS       STRING          MACRO3          COMPTYPES
    INCREMENTR      INSYMBOL        LOADADDRES      ERROR           LOAD
    EXPRESSION      GETFILENAM
    It is called by the following procedures :
    CALL


  Following procedures are declared in GETPUTRESE
  -----------------------------------------------

GETSTRINGA
    It calls the following procedures :
    LOADADDRES      NEW             ERROR           COMPTYPES       EXPRESSION
    INSYMBOL
    It is called by the following procedures :
    GETPUTRESE


  Following procedures are declared in ASSIGNMENT
  -----------------------------------------------

STOREGLOBA
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    STOREGLOBA
    It calls the following procedures :
    ERROR           STOREWORD       GETNEWGLOB
    It is called by the following procedures :
    ASSIGNMENT

  Following procedures are declared in STOREGLOBA
  -----------------------------------------------

GETNEWGLOB
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    GETNEWGLOB
    It calls the following procedures :
    NEW
    It is called by the following procedures :
    STOREGLOBA

STOREWORD
    It is the root of a strongly connected component,
        which consists of the following procedure(s) :
    STOREWORD
    It calls the following procedures :
    ERRORWITHT
    It is called by the following procedures :
    STOREGLOBA

 *******************************************************************
```